

Digital Design & Computer Arch.

Lecture 4: Combinational Logic I

Prof. Onur Mutlu

ETH Zürich

Spring 2020

28 February 2020

Assignment: Required Readings

■ This week

□ Combinational Logic

- P&P Chapter 3 until 3.3 + H&H Chapter 2

■ Next week

□ Hardware Description Languages and Verilog

- H&H Chapter 4 until 4.3 and 4.5

□ Sequential Logic

- P&P Chapter 3.4 until end + H&H Chapter 3 in full

■ By the end of next week, make sure you are done with

- **P&P Chapters 1-3 + H&H Chapters 1-4**

A Note on Hardware vs. Software

- This course might seem like it is only “Computer Hardware”
- However, you will be much more capable if you master both hardware and software (and the interface between them)
 - Can develop better software if you understand the hardware
 - Can design better hardware if you understand the software
 - Can design a better computing system if you understand both
- This course covers the HW/SW interface and microarchitecture
 - We will focus on tradeoffs and how they affect software
- Recall the four mysteries

Recap: Four Mysteries

- Meltdown & Spectre (2017-2018)
- Rowhammer (2012-2014)
- Memory Performance Attacks (2006-2007)
- Memories Forget: Refresh & RAIDR (2011-2012)

Computer Architecture as an Enabler of the Future

Assignment: Required Lecture Video

- Why study computer architecture?
- Why is it important?
- **Future Computing Architectures**
- **Required Assignment**
 - **Watch** Prof. Mutlu's inaugural lecture at ETH and understand it
 - <https://www.youtube.com/watch?v=kgiZISOcGFM>
- **Optional Assignment – for 1% extra credit**
 - **Write a 1-page summary** of the lecture and email us
 - What are your key takeaways?
 - What did you learn?
 - What did you like or dislike?
 - Submit your summary to [Moodle](#) – Deadline: April 1

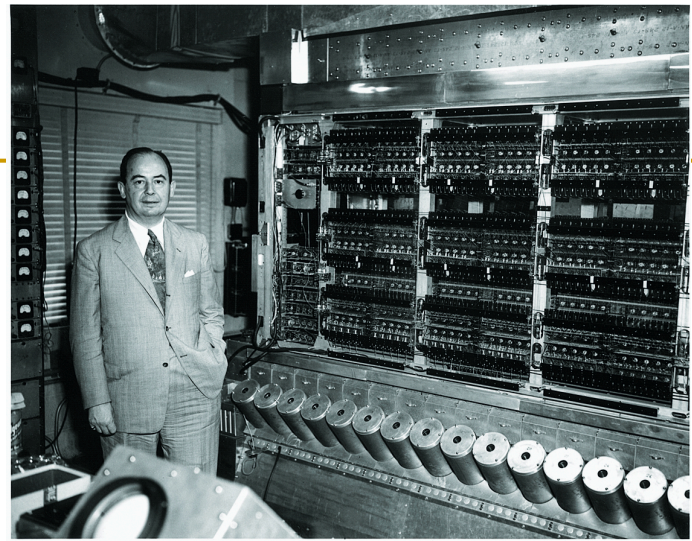
... but, first ...

- Let's understand the fundamentals...
- You can change the world only if you understand it well enough...
 - Especially the basics (fundamentals)
 - Past and present dominant paradigms
 - And, their advantages and shortcomings – tradeoffs
 - And, what remains fundamental across generations
 - And, what techniques you can use and develop to solve problems

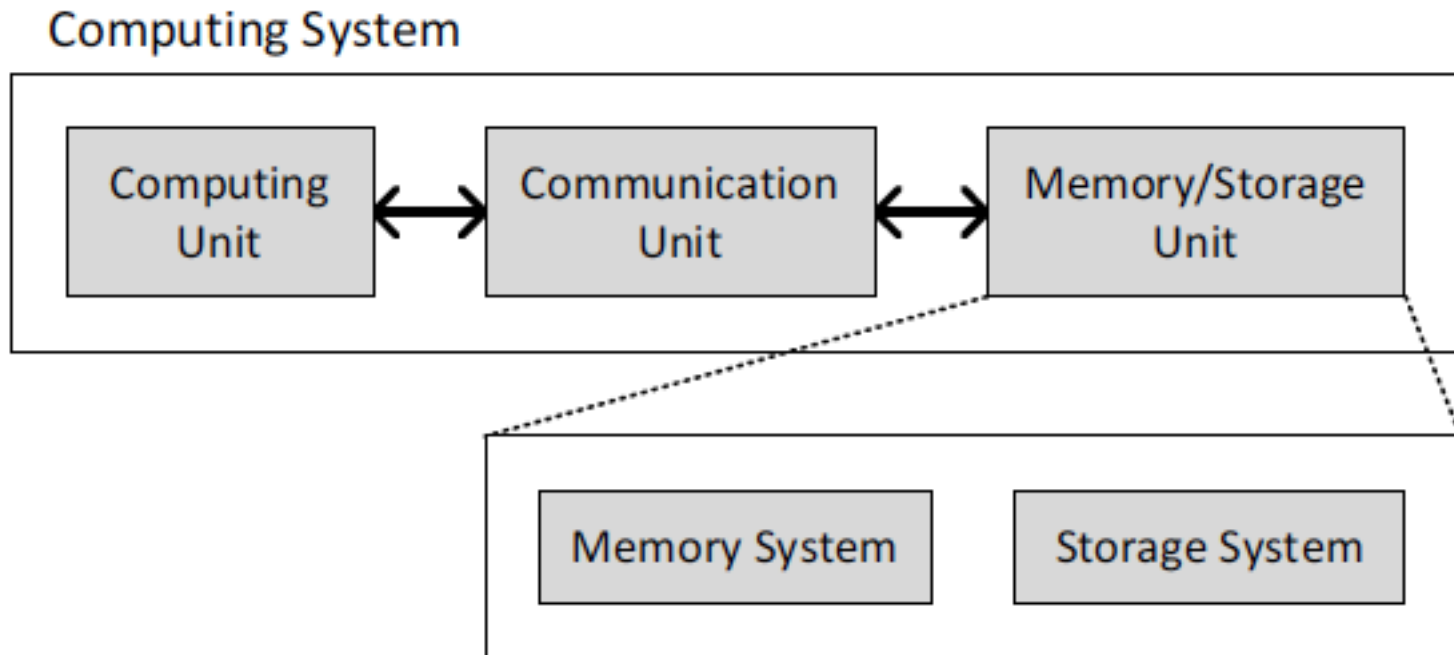
Fundamental Concepts

What is A Computer?

- Three key components
- Computation
- Communication
- Storage/memory

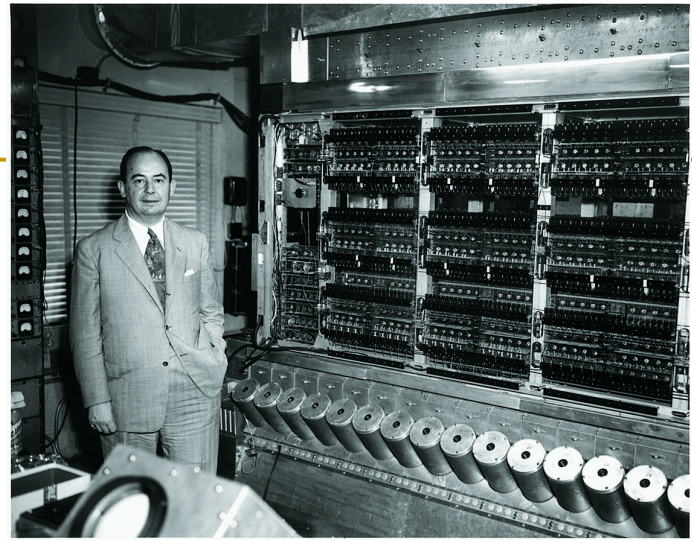


Burks, Goldstein, von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," 1946.



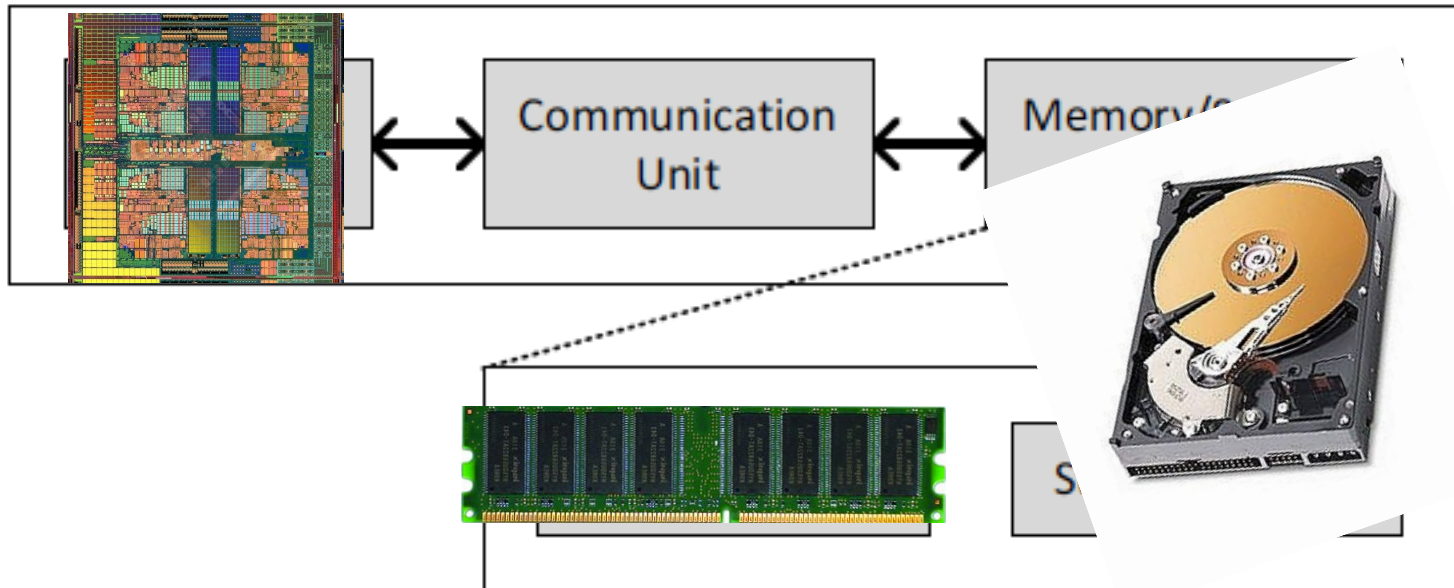
What is A Computer?

- Three key components
- Computation
- Communication
- Storage/memory



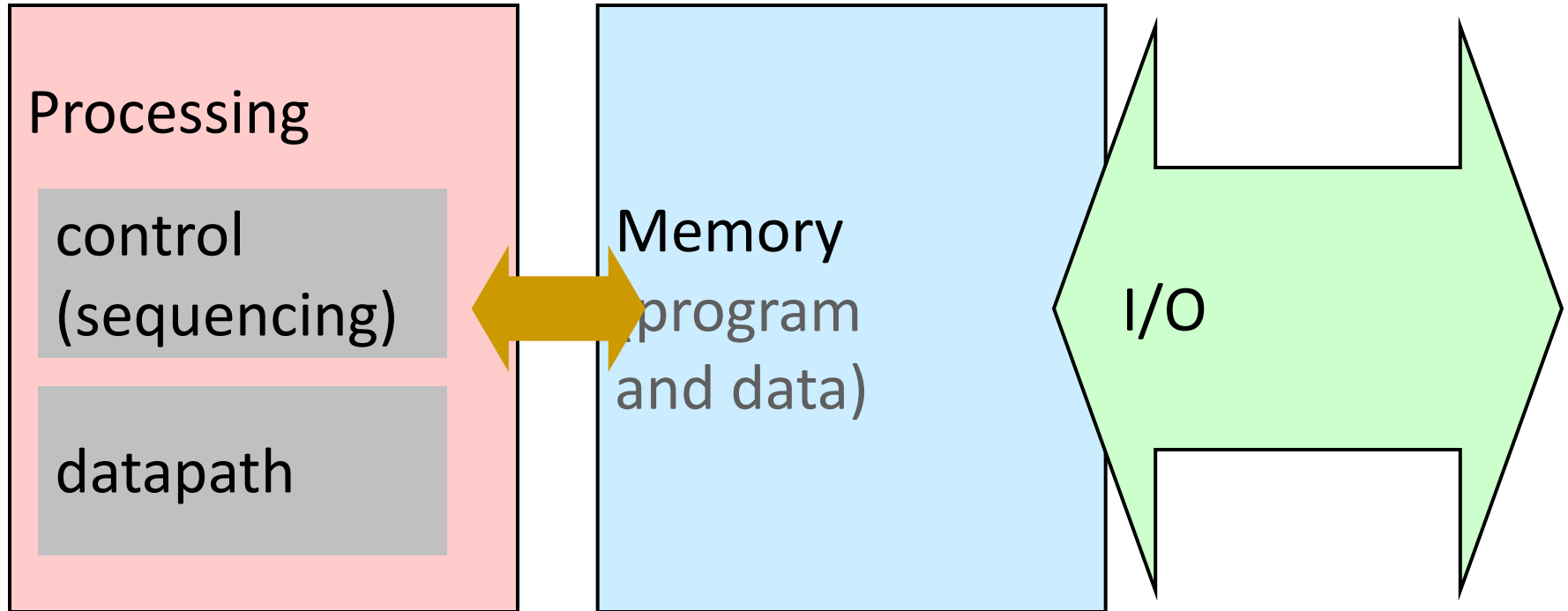
Burks, Goldstein, von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," 1946.

Computing System

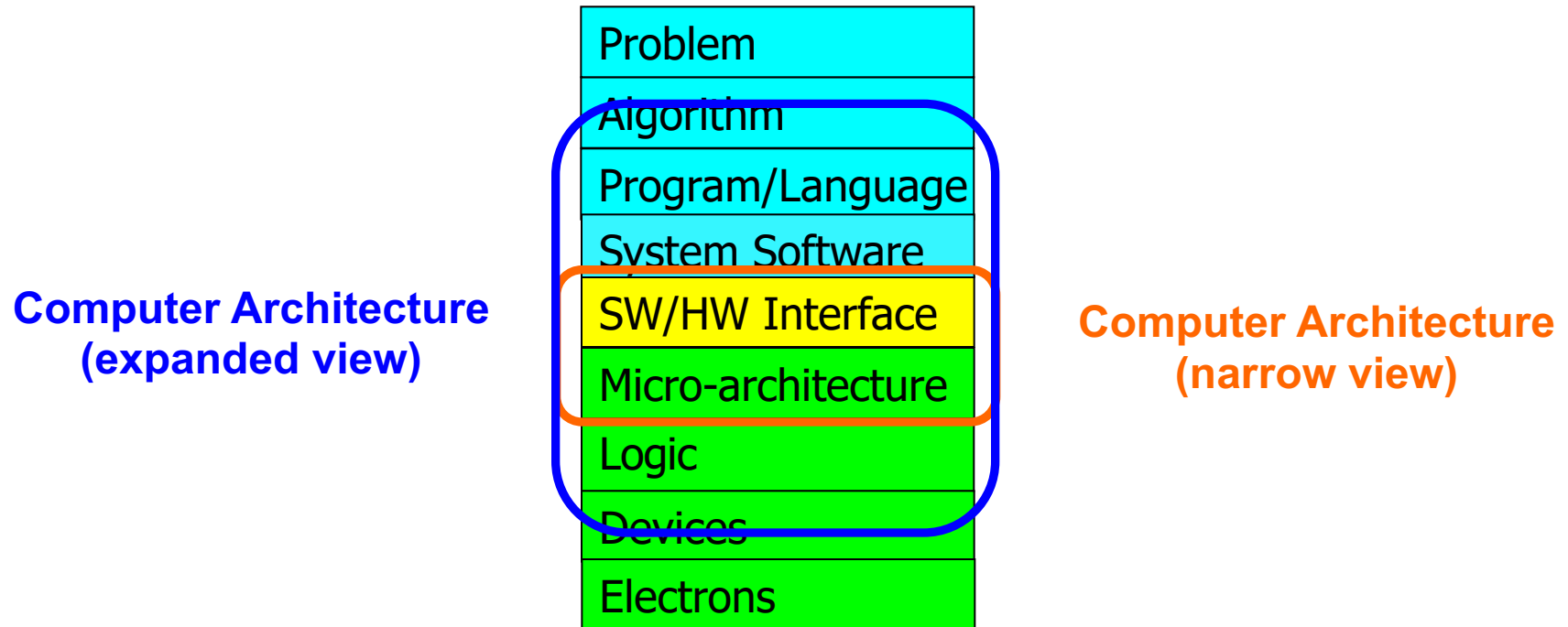


What is A Computer?

- We will cover all three components



Recall: The Transformation Hierarchy



What We Will Cover (I)

- Combinational Logic Design
- Hardware Description Languages (Verilog)
- Sequential Logic Design
- Timing and Verification
- ISA (MIPS and LC3b)
- MIPS Assembly Programming

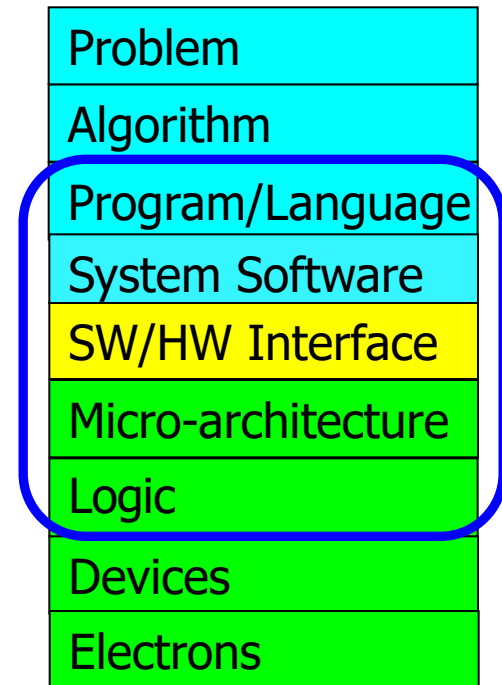
Problem
Algorithm
Program/Language
System Software
SW/HW Interface
Micro-architecture
Logic
Devices
Electrons

What We Will Cover (II)

- Microarchitecture Basics: Single-cycle
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Other Processing Paradigms (SIMD, VLIW, Systolic, ...)
- Memory and Caches
- Virtual Memory

Processing Paradigms We Will Cover

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- SIMD Processing (Vector & array, GPUs)
- Decoupled Access Execute
- Systolic Arrays

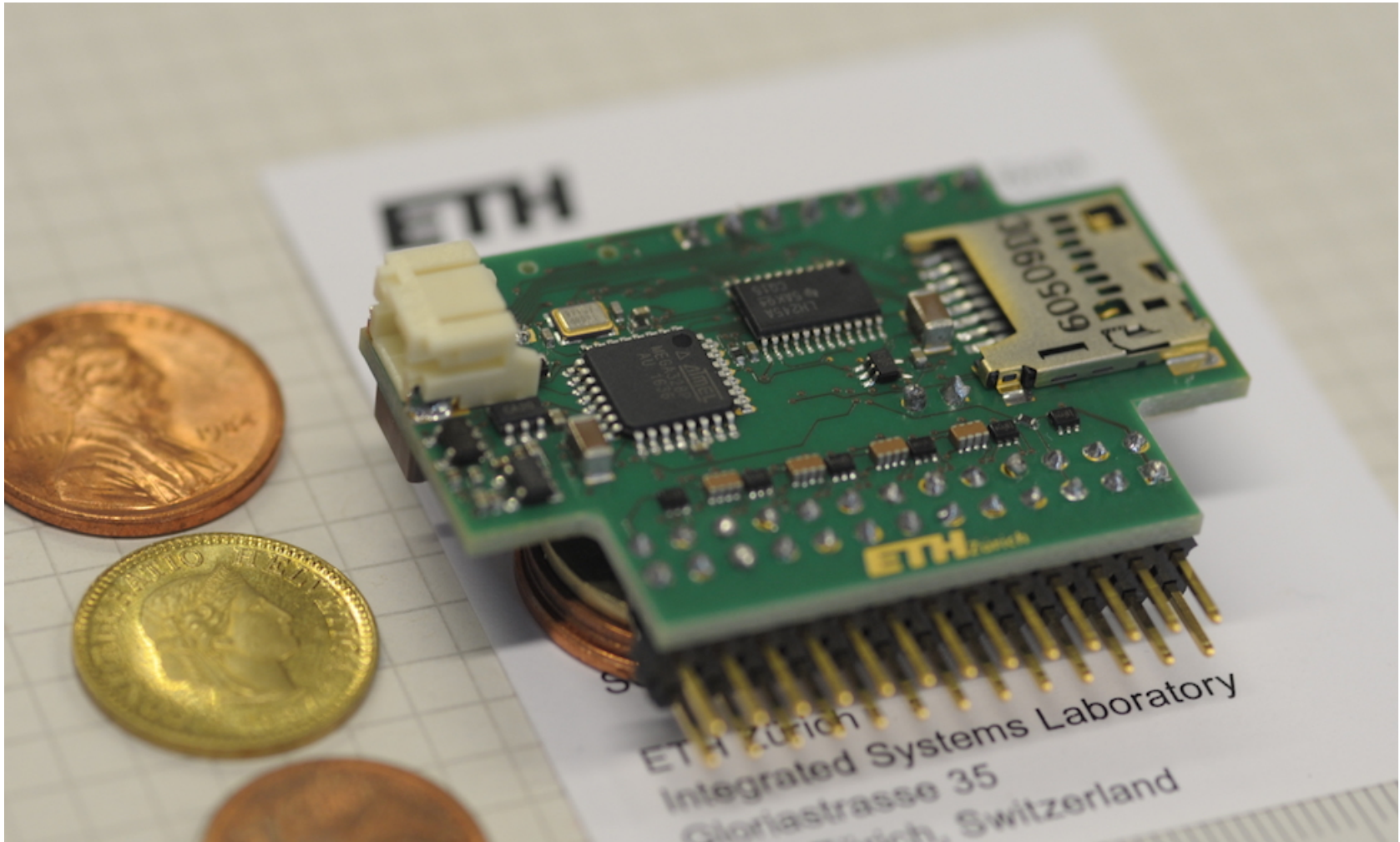


Combinational Logic Circuits and Design

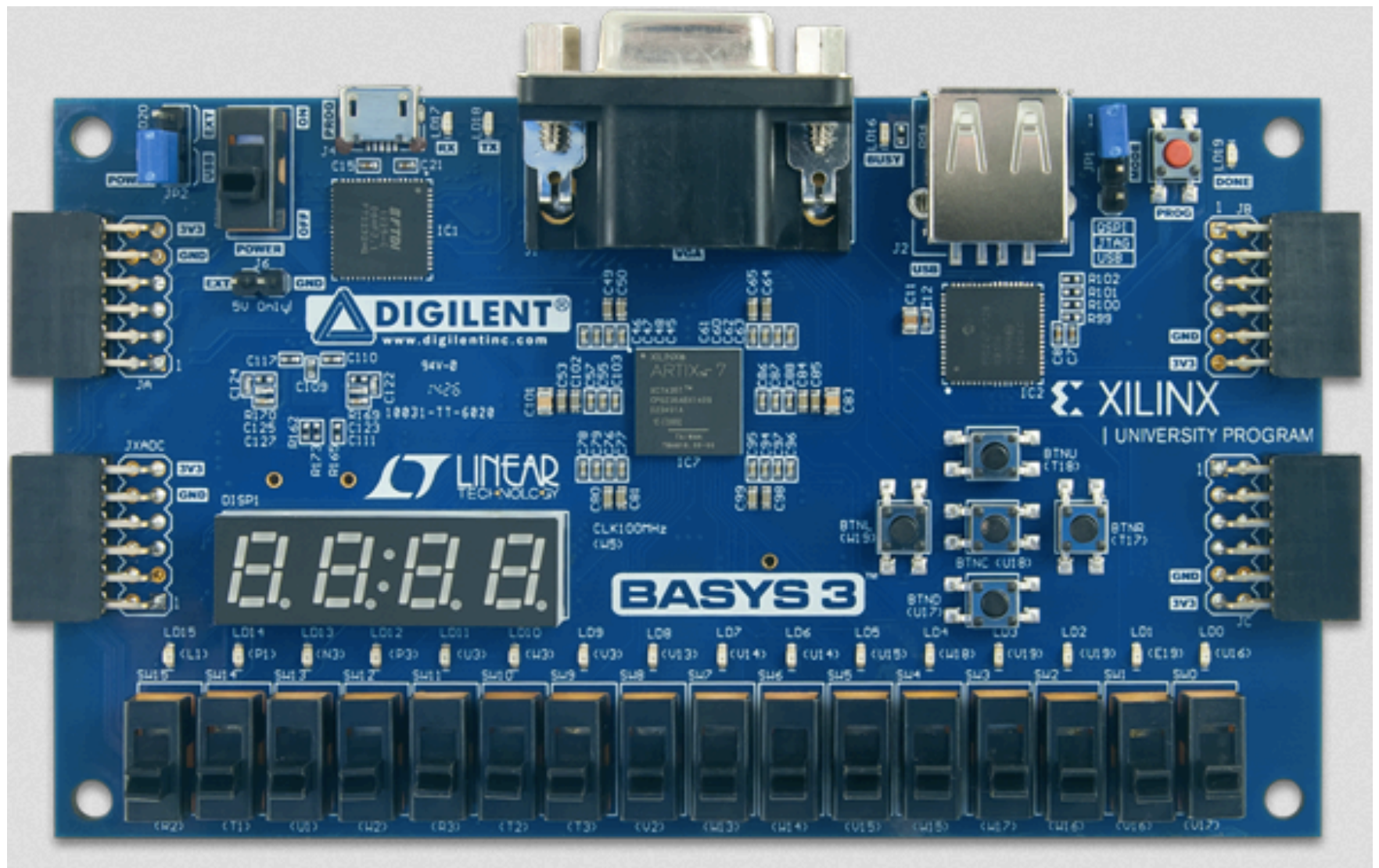
What Will We Learn Today?

- Building blocks of modern computers
 - Transistors
 - Logic gates
- Boolean algebra
- Combinational circuits
- How to use Boolean algebra to represent combinational circuits
- Minimizing logic circuits (if time permits)

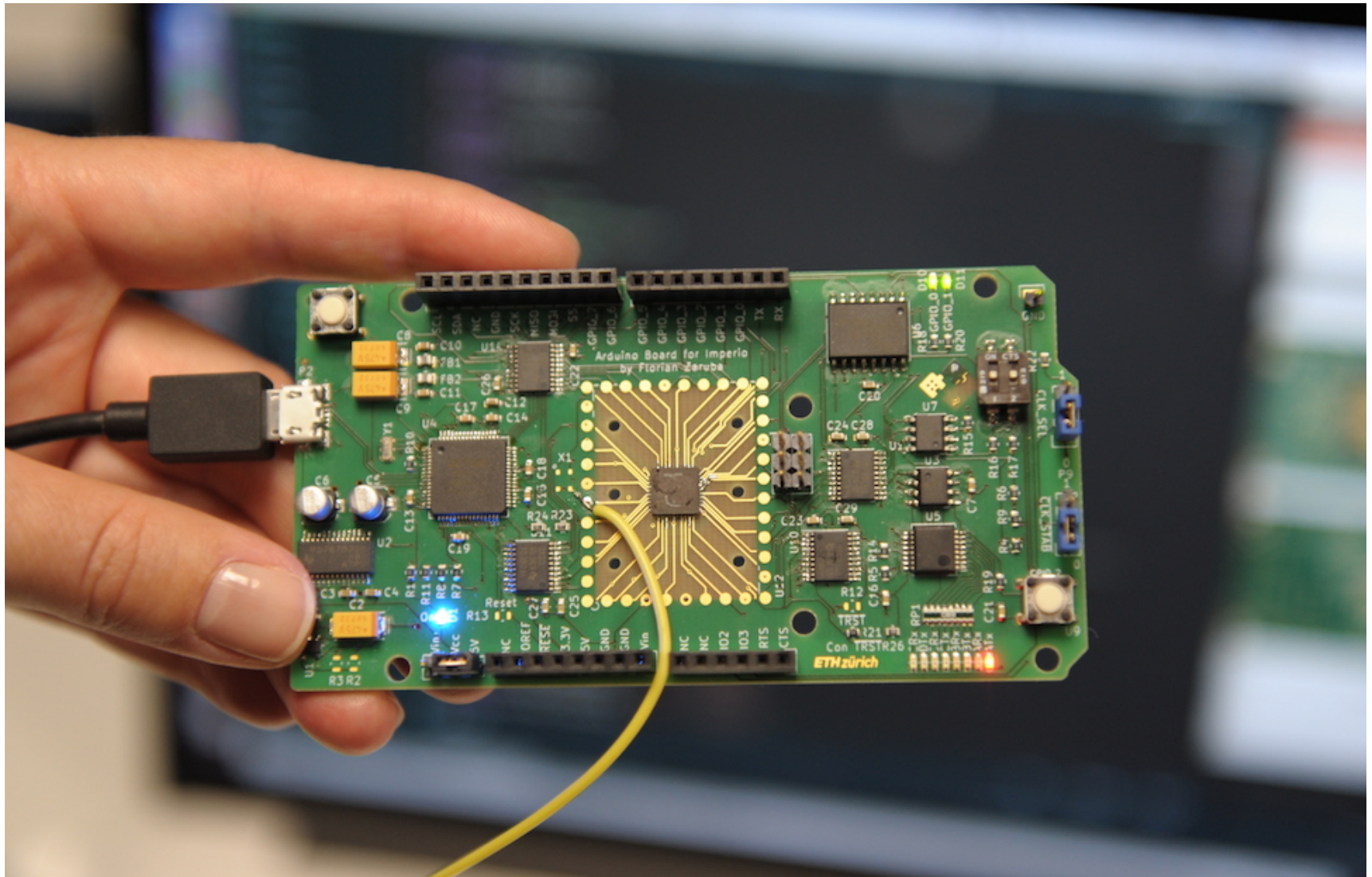
(Micro)-Processors






FPGAs






Custom ASICs






They All Look the Same

	Microprocessors	FPGAs	ASICs
			
In short:	Common building block of computers	Reconfigurable hardware, flexible	You customize everything




They All Look the Same

	Microprocessors	FPGAs	ASICs
			
In short:	Common building block of computers	Reconfigurable hardware, flexible	You customize everything
Program Development Time	minutes	days	months




They All Look the Same

	Microprocessors	FPGAs	ASICs
			
In short:	Common building block of computers	Reconfigurable hardware, flexible	You customize everything
Program Development Time	minutes	days	months
Performance	0	+	++



They All Look the Same

	Microprocessors	FPGAs	ASICs
			
In short:	Common building block of computers	Reconfigurable hardware, flexible	You customize everything
Program Development Time	minutes	days	months
Performance	0	+	++
Good for	Ubiquitous Simple to use	Prototyping Small volume	Mass production, Max performance

They All Look the Same

	Microprocessors	FPGAs	ASICs
			
In short:	Common building block of computers	Reconfigurable hardware, flexible	You customize everything
Program Development Time	minutes	days	months
Performance	0	+	++
Good for	Ubiquitous Simple to use	Prototyping Small volume	Mass production, Max performance
Programming	Executable file	Bit file	Design masks
Languages	C/C++/Java/...	Verilog/VHDL	Verilog/VHDL
Main Companies	Intel, ARM, AMD	Xilinx, Altera, Lattice	TSMC, UMC, ST, Globalfoundries

They All Look the Same

	Microprocessors	FPGAs	
	<div>Want to learn how these work</div> 	<div>By programming these</div> 	
In short	Common building block of computers	Reconfigurable hardware, flexible	Optimize
Program Development Time	minutes	days	months
Performance	0	+	++
Good for	Ubiquitous Simple to use	Prototyping	Mass production.
Programming Languages	Executable file C/C++/Java/...	Using this language	
Main Companies	Intel, ARM, AMD	Verilog/VHDL Xilinx, Altera, Lattice	Verilog/VHDL TSMC, UMC, ST, Globalfoundries

Building Blocks of Modern Computers

Transistors

Transistors

■ Computers are built from very large numbers of very simple structures

- ❑ Intel's Pentium IV microprocessor, first offered for sale in 2000, was made up of more than **42 million MOS transistors**
- ❑ Intel's Core i7 Broadwell-E, offered for sale in 2016, is made up of more than **3.2 billion MOS transistors**

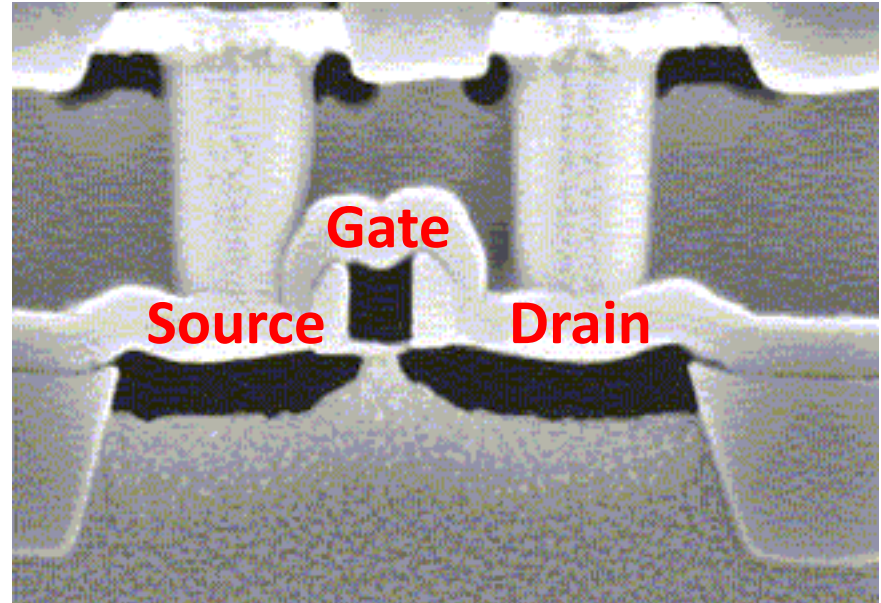
■ This lecture

- ❑ How the MOS transistor works (as a logic element)
- ❑ How these transistors are connected to form logic gates
- ❑ How logic gates are interconnected to form larger units that are needed to construct a computer

Problem
Algorithm
Program/Language
Runtime System (VM, OS, MM)
ISA (Architecture)
Microarchitecture
Logic
Devices
Electrons

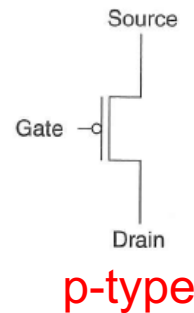
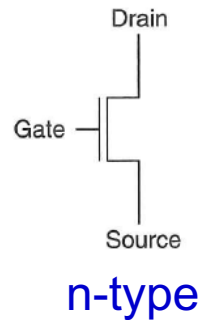
MOS Transistor

- By combining
 - Conductors (**M**etal)
 - Insulators (**O**xide)
 - **S**emiconductors
- We get a Transistor (MOS)
- Why is this useful?
 - We can combine many of these to realize simple logic gates
- The **electrical properties** of metal-oxide semiconductors are well **beyond** the scope of what we want to understand in this course
 - They are below our lowest level of abstraction



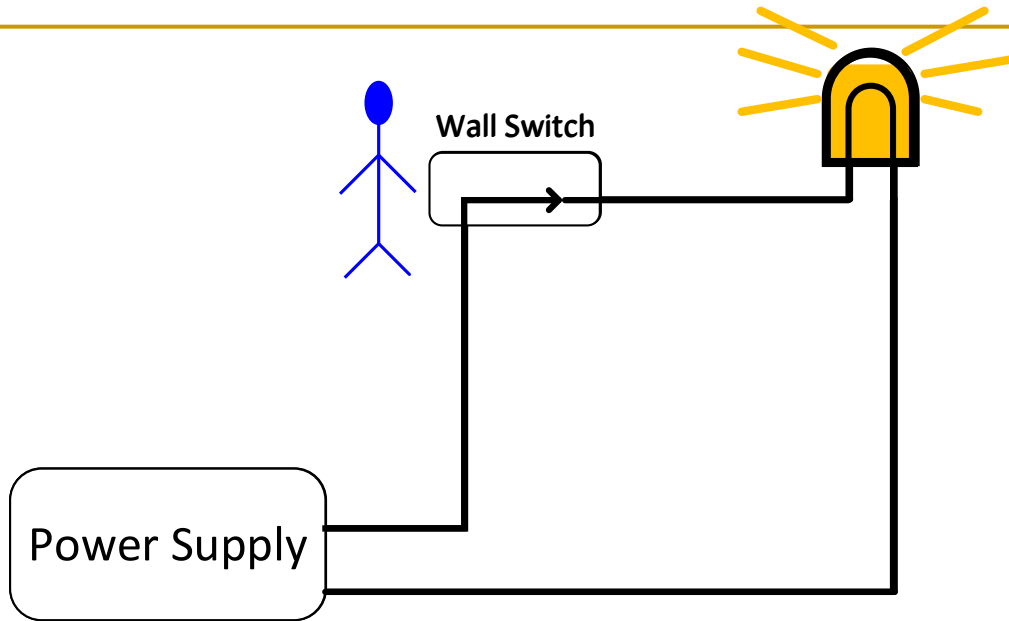
Different Types of MOS Transistors

- There are two types of MOS transistors: n-type and p-type



- They both operate “logically,” very similar to the way wall switches work

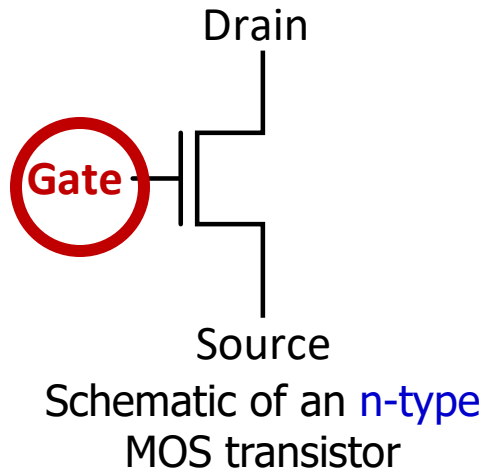
How Does a Transistor Work?



- ❑ In order for the lamp to glow, **electrons must flow**
- ❑ In order for electrons to flow, there must be a **closed circuit** from the power supply to the lamp and back to the power supply
- ❑ The lamp can be **turned on and off** by simply manipulating the wall switch to make or break the closed circuit

How Does a Transistor Work?

- Instead of the wall switch, we could use an **n-type** or a **p-type** MOS transistor to make or break the closed circuit



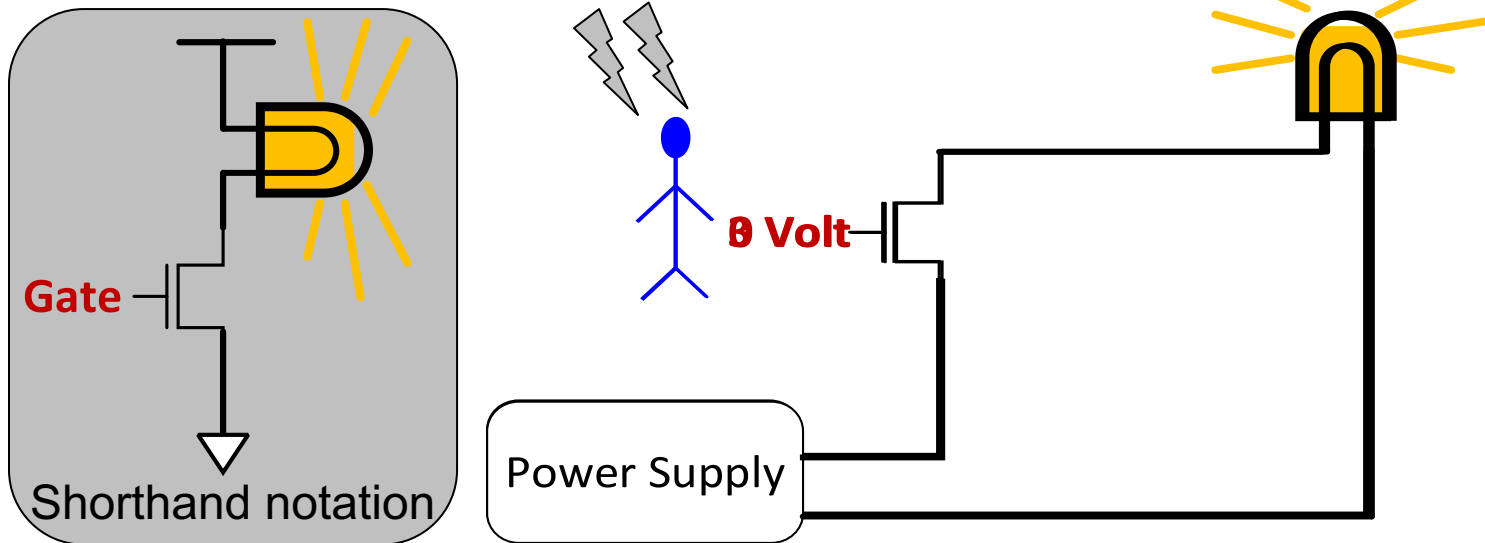
If the gate of an **n-type** transistor is supplied with a **high** voltage, the connection from source to drain acts like a piece of wire

Depending on the technology, 0.3V to 3V

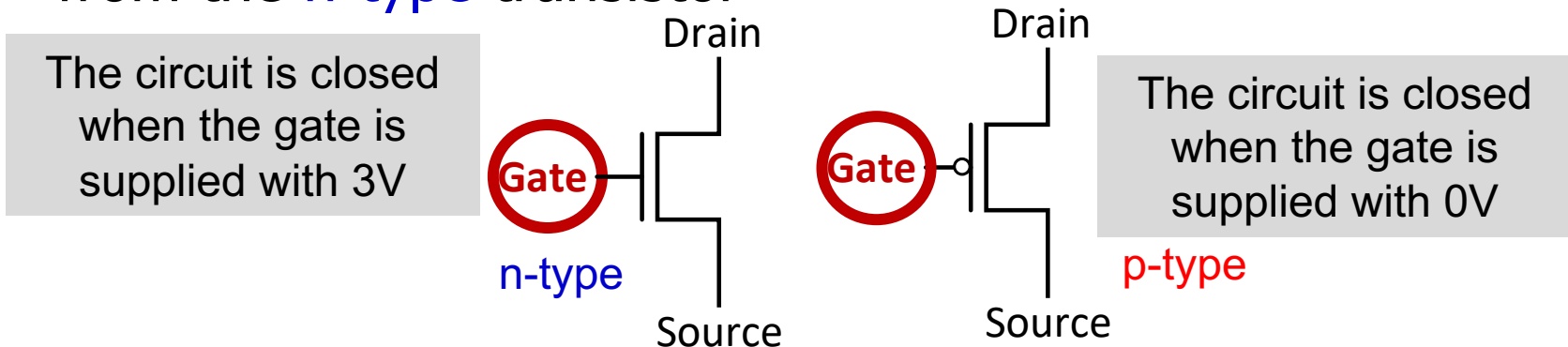
If the gate of the n-type transistor is supplied with 0V, the connection between the source and drain is broken

How Does a Transistor Work?

- The **n-type** transistor in a circuit with a battery and a bulb



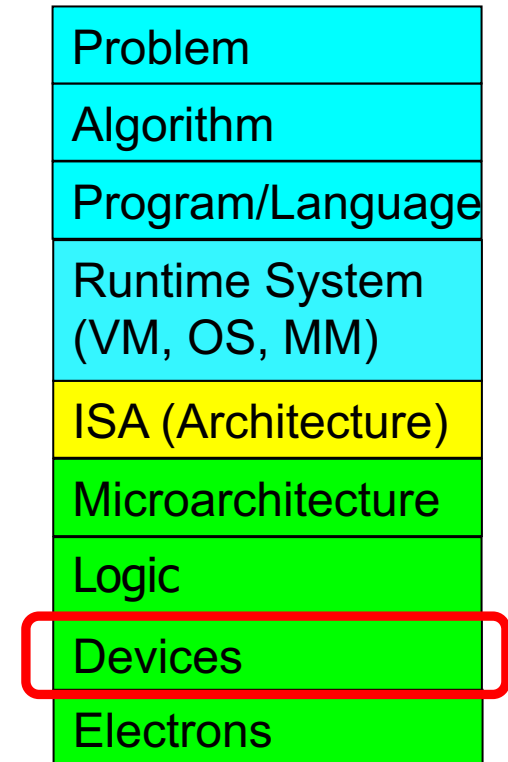
- The **p-type** transistor works in exactly the opposite fashion from the **n-type** transistor



Logic Gates

One Level Higher in the Abstraction

- **Now, we know how a MOS transistor works**
- How do we build logic out of MOS transistors?
- We construct basic logic structures out of individual MOS transistors
- These **logical units** are named **logic gates**
 - They implement simple **Boolean** functions

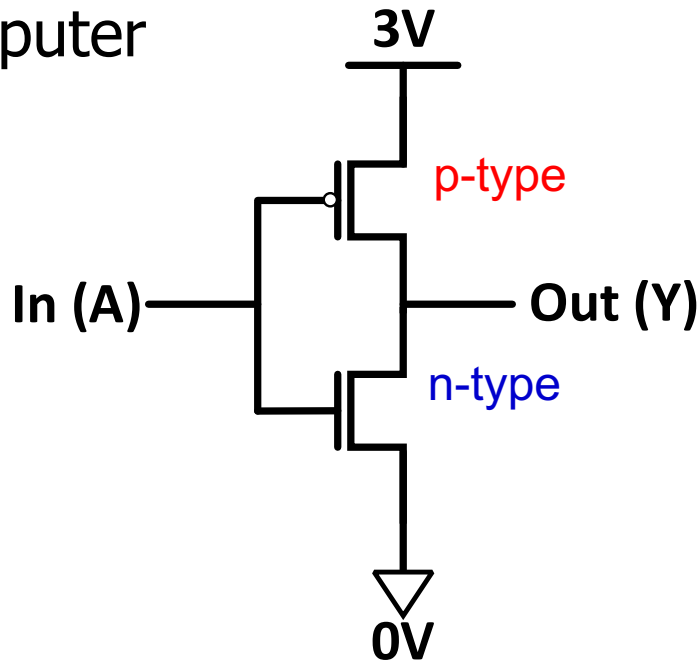


Making Logic Blocks Using CMOS Technology

- Modern computers use both **n-type** and **p-type** transistors, i.e. Complementary MOS (**CMOS**) technology

$$\text{nMOS} + \text{pMOS} = \text{CMOS}$$

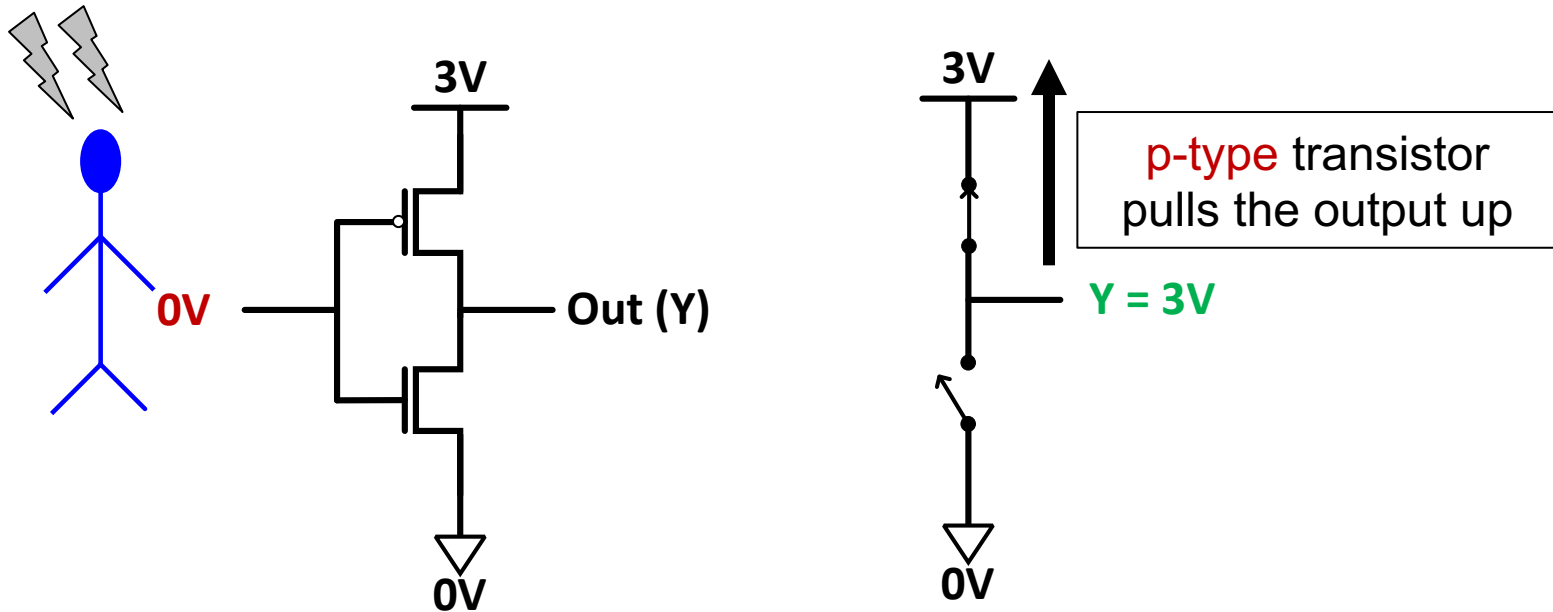
- The simplest logic structure that exists in a modern computer



What does this circuit do?

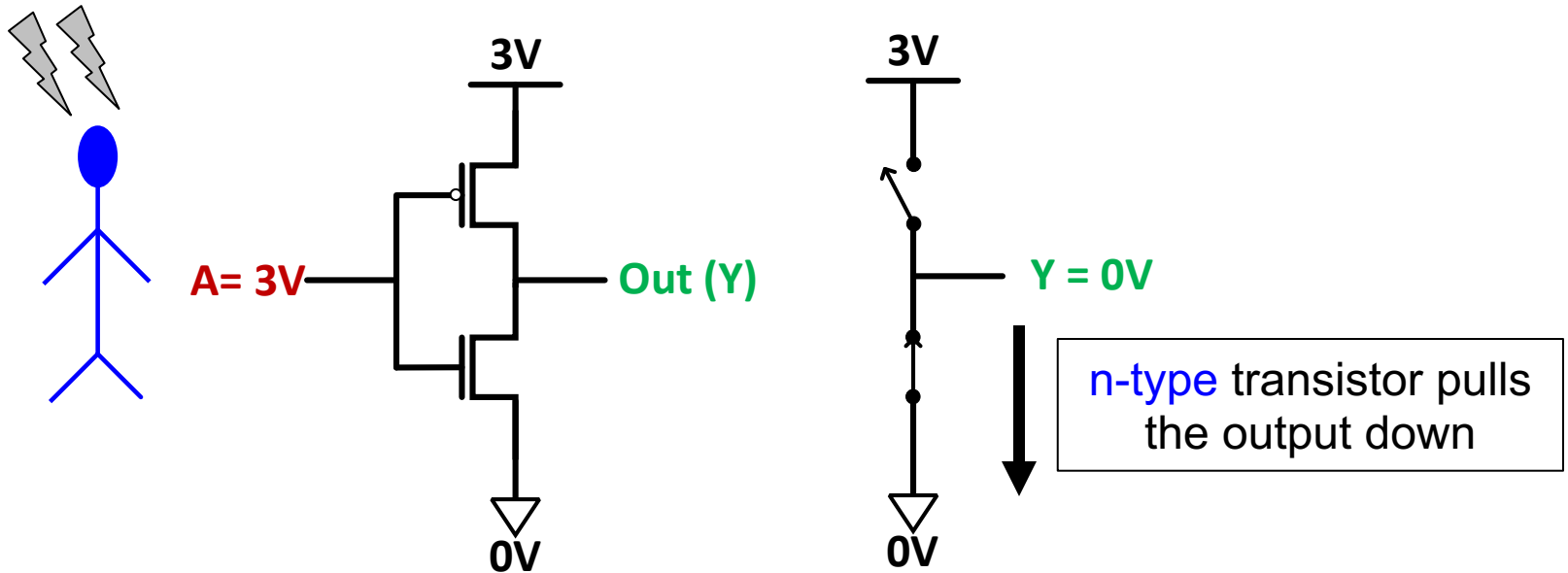
Functionality of Our CMOS Circuit

What happens when the input is connected to 0V?



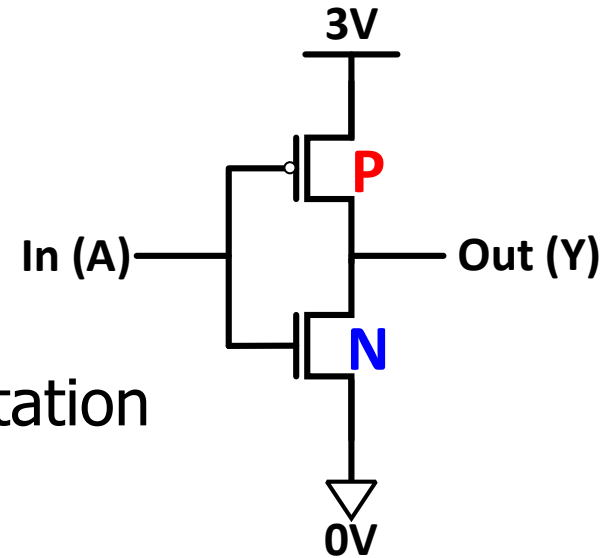
Functionality of Our CMOS Circuit

What happens when the input is connected to 3V?



CMOS NOT Gate

- This is actually the **CMOS NOT Gate**
- Why do we call it NOT?
 - If $A = 0V$ then $Y = 3V$
 - If $A = 3V$ then $Y = 0V$
- **Digital circuit:** one possible interpretation
 - Interpret **0V** as logical (binary) **0** value
 - Interpret **3V** as logical (binary) **1** value

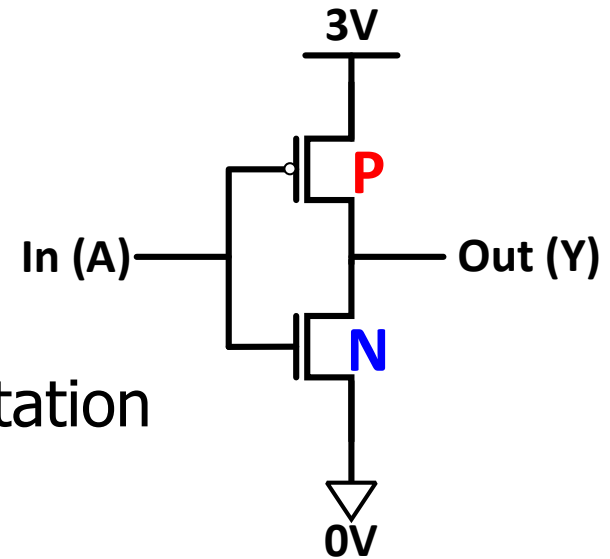


A	P	N	Y
0	ON	OFF	1
1	OFF	ON	0

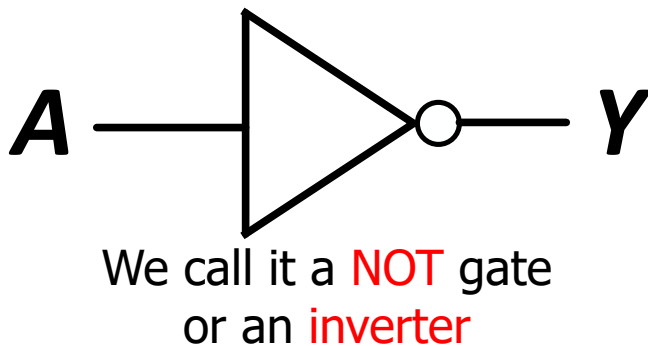
$$Y = \bar{A}$$

CMOS NOT Gate

- This is actually the CMOS NOT Gate
- Why do we call it NOT?
 - If $A = 0V$ then $Y = 3V$
 - If $A = 3V$ then $Y = 0V$
- **Digital circuit:** one possible interpretation
 - Interpret $0V$ as logical (binary) 0 value
 - Interpret $3V$ as logical (binary) 1 value



$$Y = \bar{A}$$

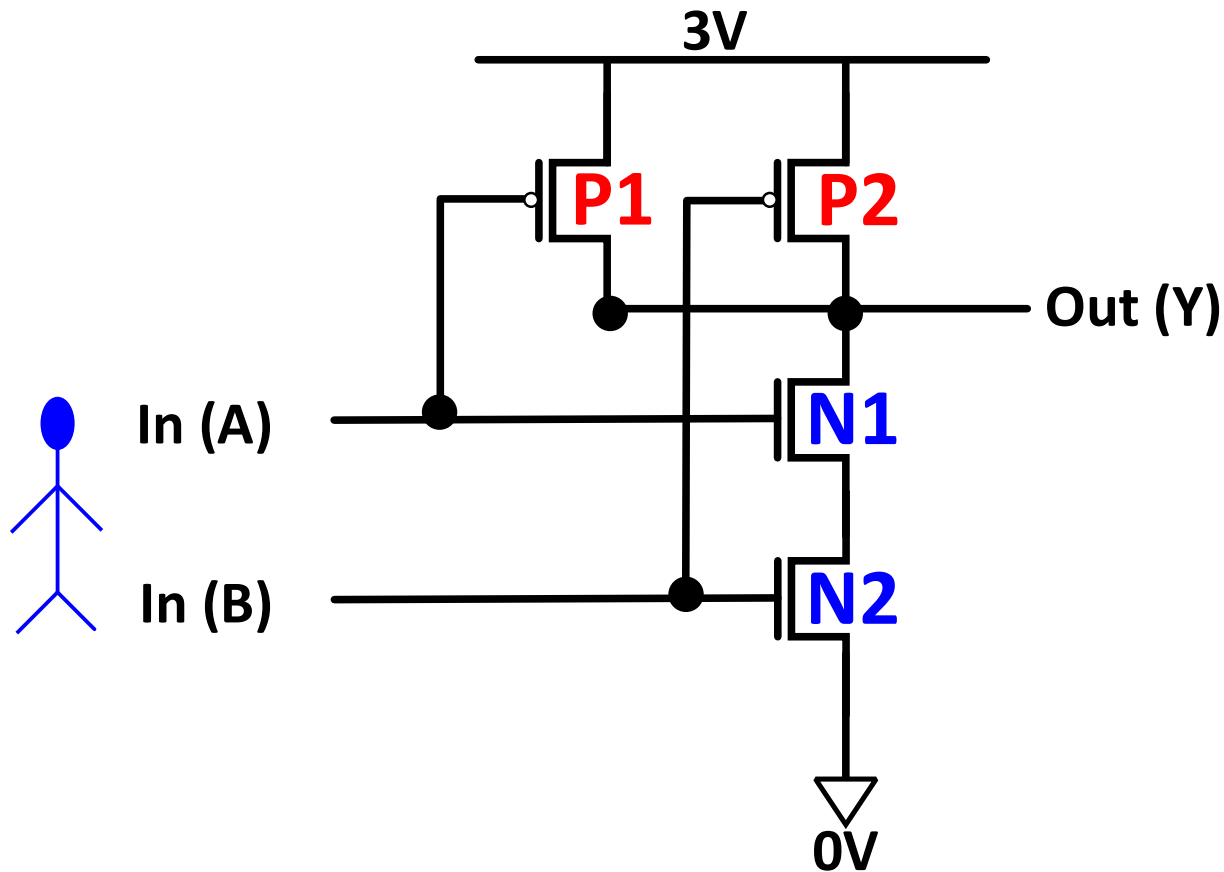


Truth table: what would be the logical output of the circuit for each possible input

A	Y
0	1
1	0

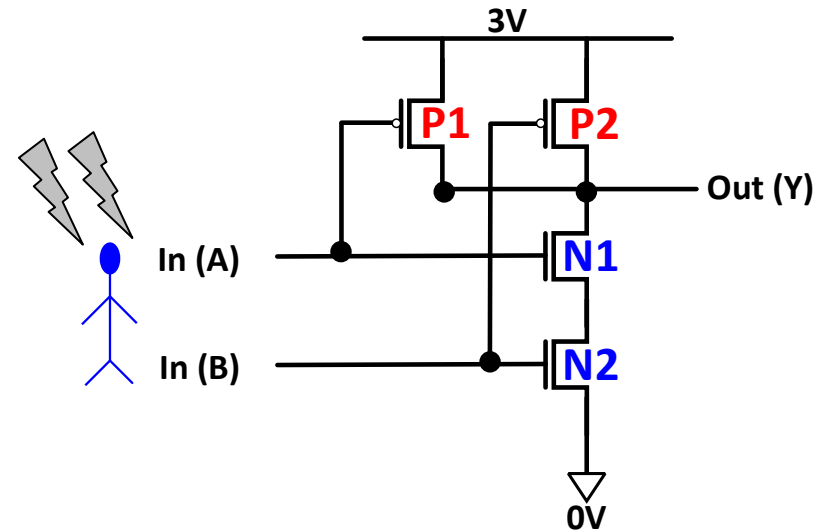
Another CMOS Gate: What Is This?

- Let's build more complex gates!



CMOS NAND Gate

- Let's build more complex gates!



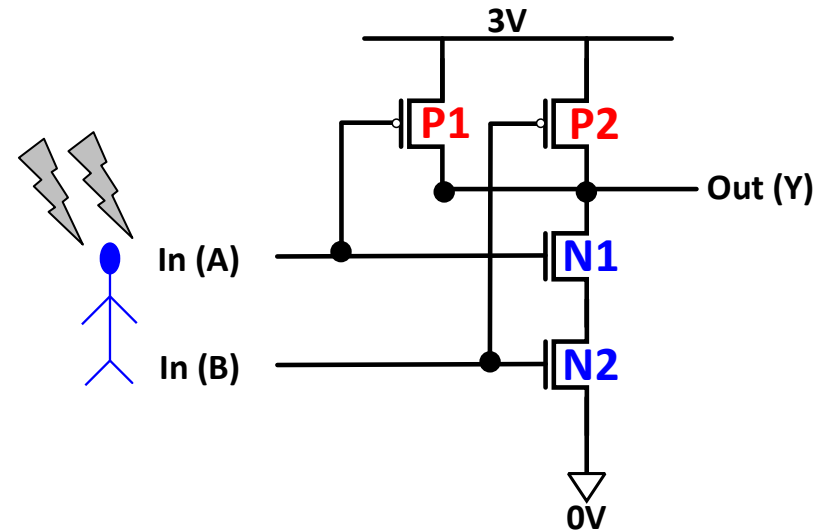
$$Y = \overline{A \cdot B} = \overline{AB}$$

A	B	P1	P2	N1	N2	Y
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	1
1	0	OFF	ON	ON	OFF	1
1	1	OFF	OFF	ON	ON	0

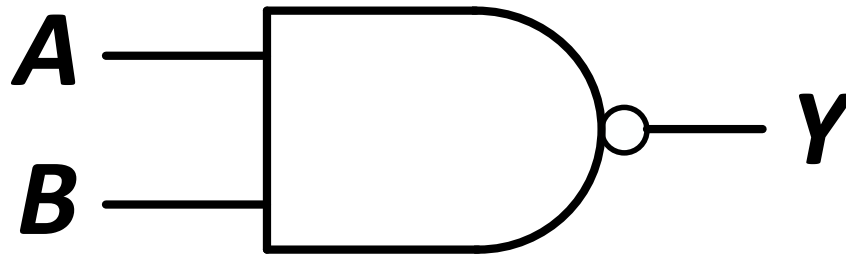
- P1 and P2 are in **parallel**; only one must be ON to pull the output up to 3V
- N1 and N2 are connected in **series**; both must be ON to pull the output to 0V

CMOS NAND Gate

- Let's build more complex gates!



$$Y = \overline{A \cdot B} = \overline{AB}$$



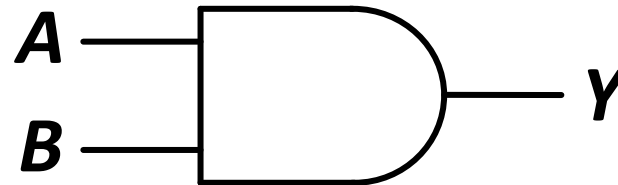
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

CMOS AND Gate

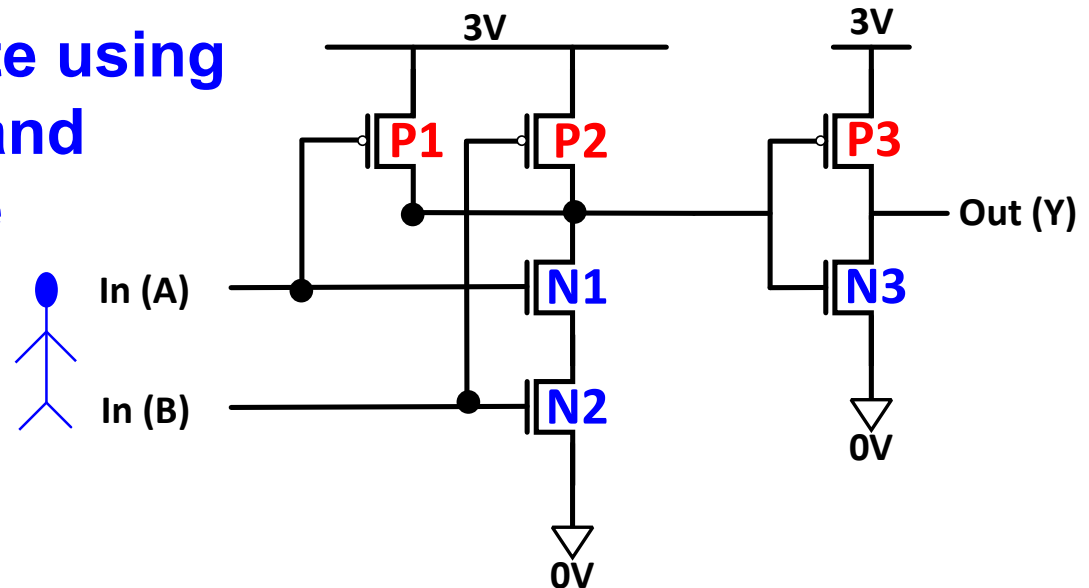
- How can we make an AND gate?

<i>A</i>	<i>B</i>	<i>Y</i>
0	0	0
0	1	0
1	0	0
1	1	1

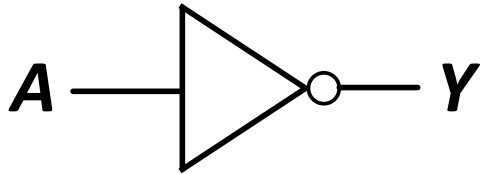
$$Y = A \cdot B = AB$$



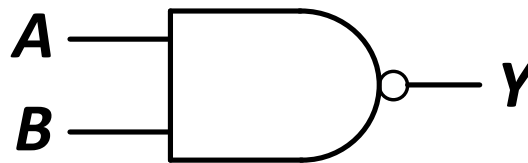
We make an **AND** gate using
one **NAND** gate and
one **NOT** gate



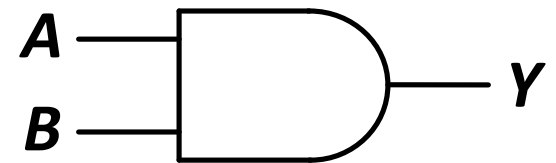
CMOS NOT, NAND, AND Gates



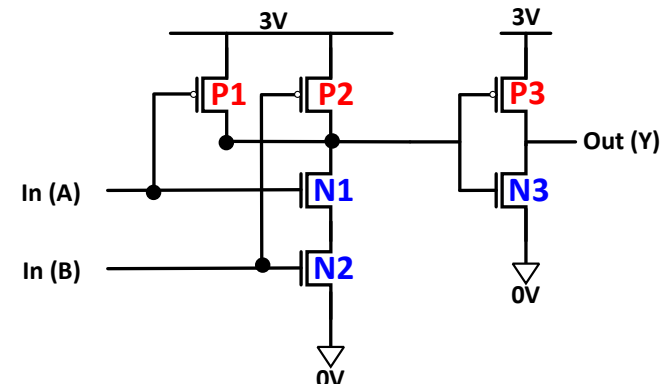
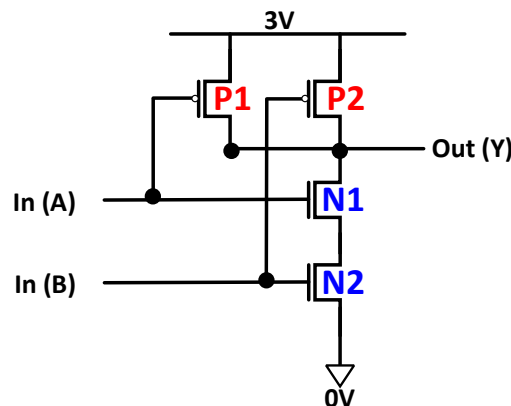
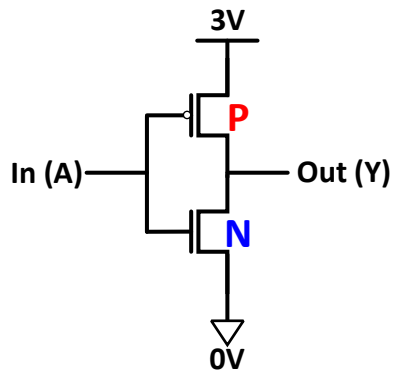
A	Y
0	1
1	0



A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

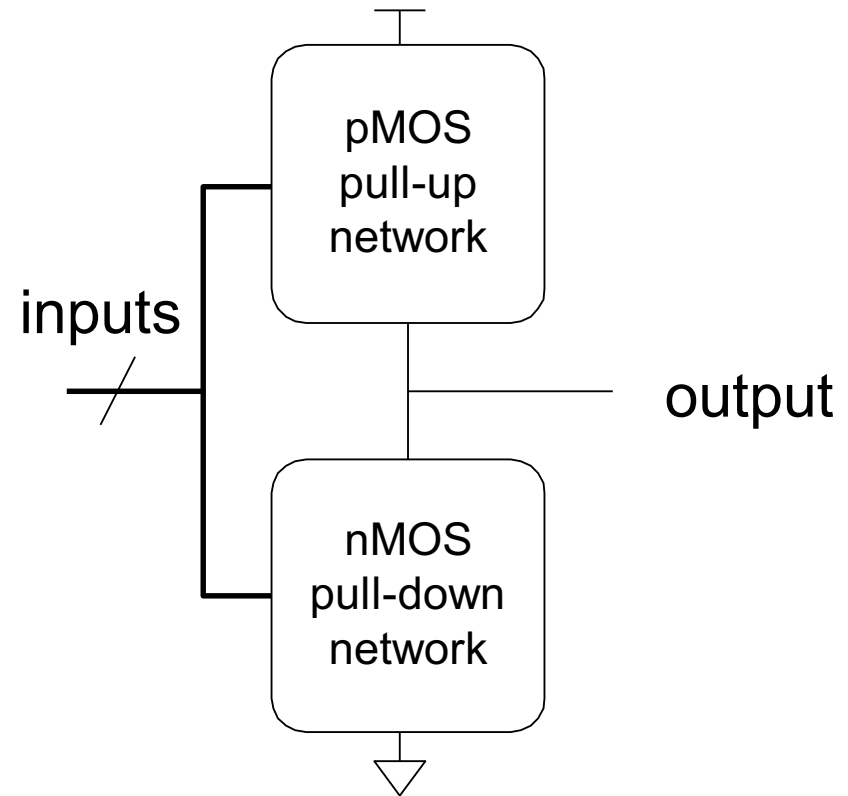


A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



General CMOS Gate Structure

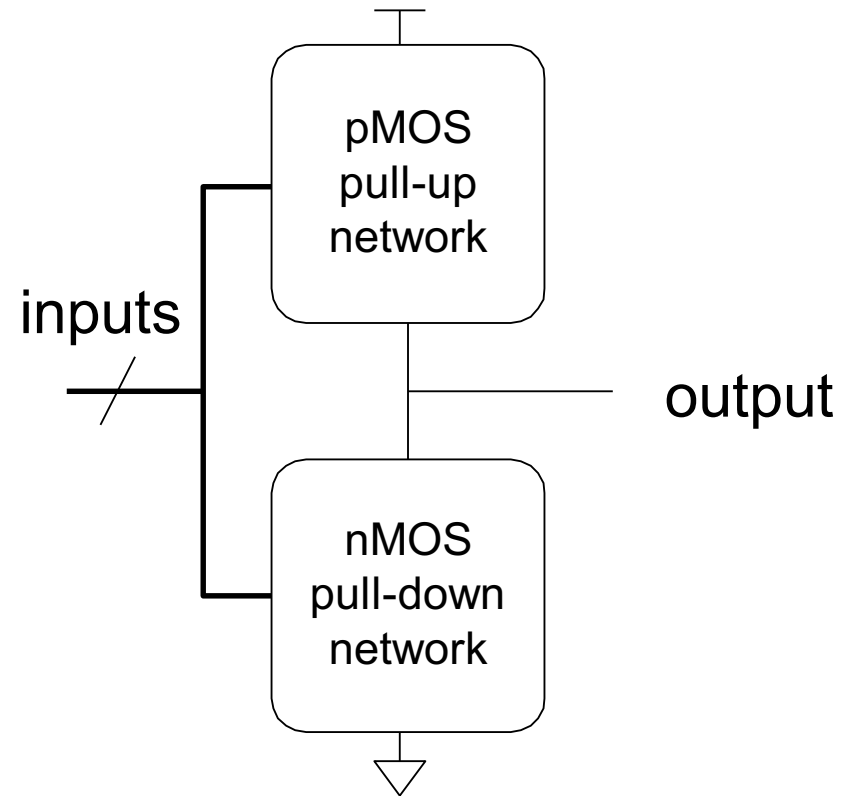
- The general form used to construct any inverting logic gate, such as: NOT, NAND, or NOR
 - The networks may consist of transistors in series or in parallel
 - When transistors are in parallel, the network is **ON** if one of the transistors is **ON**
 - When transistors are in series, the network is **ON** only if all transistors are **ON**



pMOS transistors are used for pull-up
nMOS transistors are used for pull-down

General CMOS Gate Structure (II)

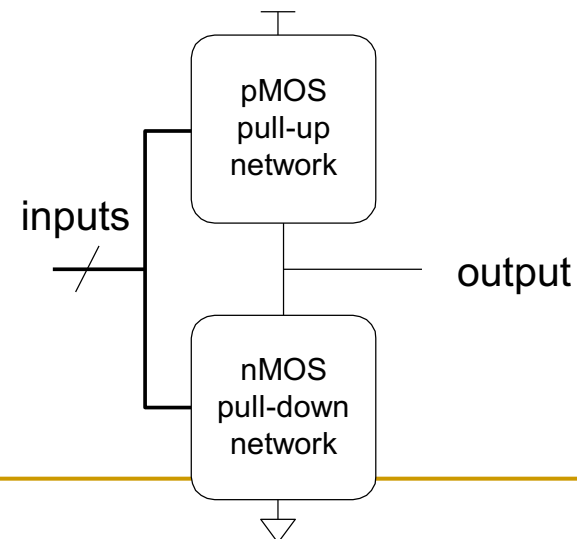
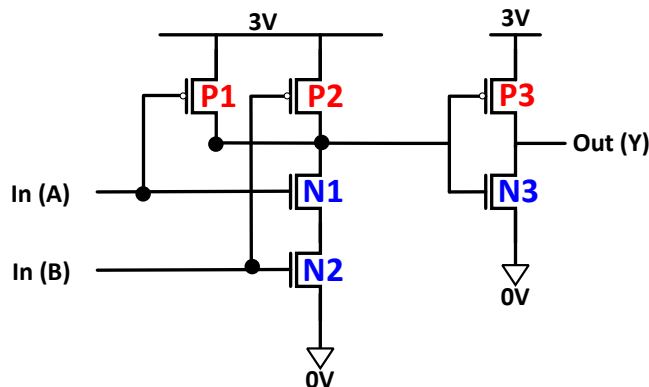
- Exactly one network should be ON, and the other network should be OFF at any given time
 - If both networks are ON at the same time, there is a **short circuit** → likely incorrect operation
 - If both networks are OFF at the same time, the output is **floating** → undefined



pMOS transistors are used for pull-up
nMOS transistors are used for pull-down

Digging Deeper: Why This Structure?

- MOS transistors are **not perfect** switches
- pMOS transistors pass 1's well but 0's poorly
- nMOS transistors pass 0's well but 1's poorly
- pMOS transistors are good at "pulling up" the output
- nMOS transistors are good at "pulling down" the output



Digging Deeper: Latency

- Which one is faster?
 - Transistors in series
 - Transistors in parallel
- Series connections are slower than parallel connections
 - More resistance on the wire
- How do you alleviate this latency?
 - See H&H Section 1.7.8 for an example: [pseudo-nMOS Logic](#)

Digging Deeper: Power Consumption

- Dynamic Power Consumption

- $C * V^2 * f$

- C = capacitance of the circuit (wires and gates)
 - V = supply voltage
 - f = charging frequency of the capacitor

- Static Power consumption

- $V * I_{\text{leakage}}$

- supply voltage * leakage current

- Energy Consumption

- Power * Time

- See more in H&H Chapter 1.8

Common Logic Gates

Buffer



A	Z
0	0
1	1

AND



A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

OR



A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

XOR



A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

Inverter



A	Z
0	1
1	0

NAND



A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

NOR



A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

XNOR



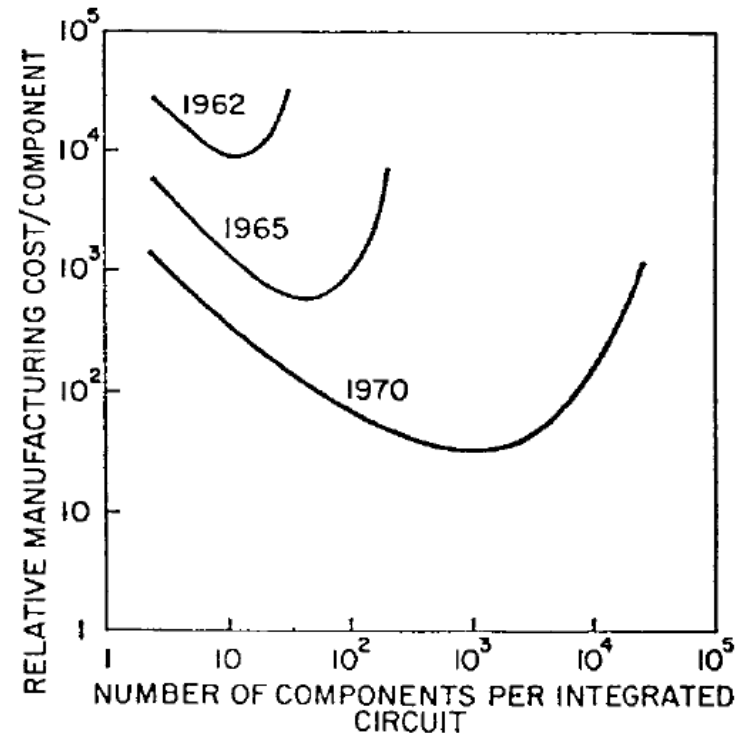
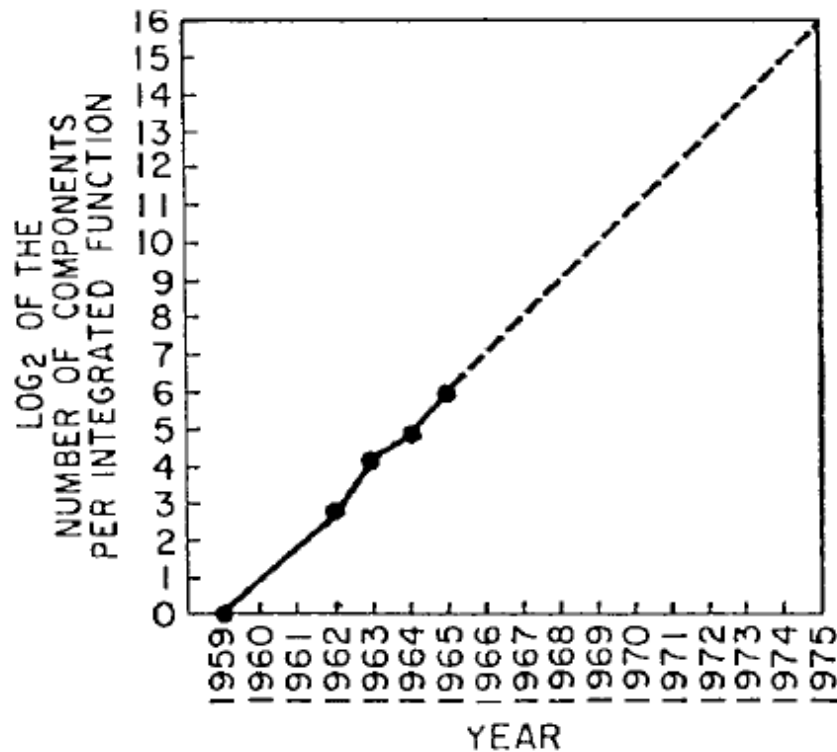
A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

Larger Gates

- We can extend the gates to more than 2 inputs
- Example: 3-input AND gate, 10-input NOR gate
- See your readings

Aside: Moore's Law: Enabler of Many Gates on a Chip

An Enabler: Moore's Law

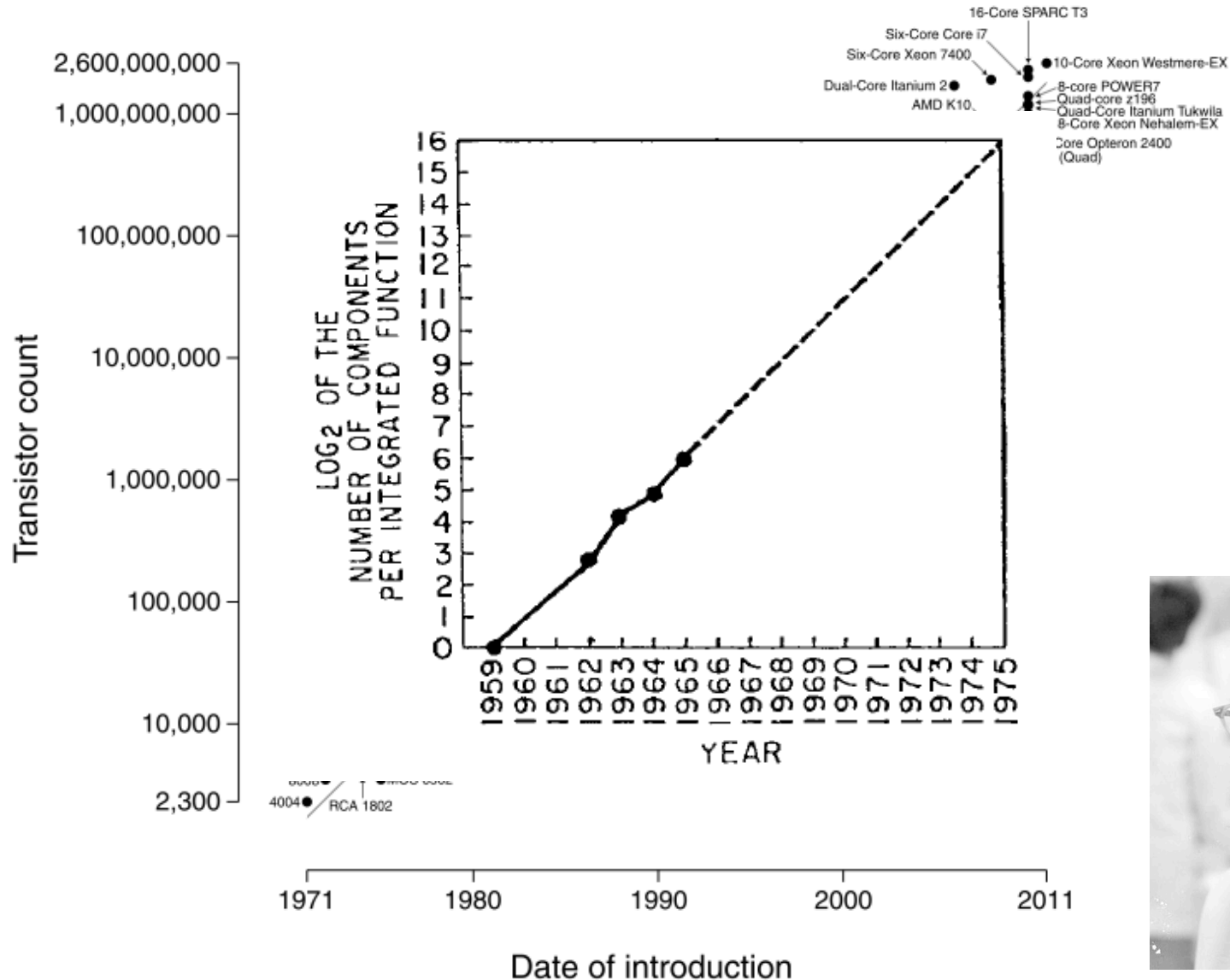


Moore, “Cramming more components onto integrated circuits,”

Electronics Magazine, 1965.

Component counts double every other year

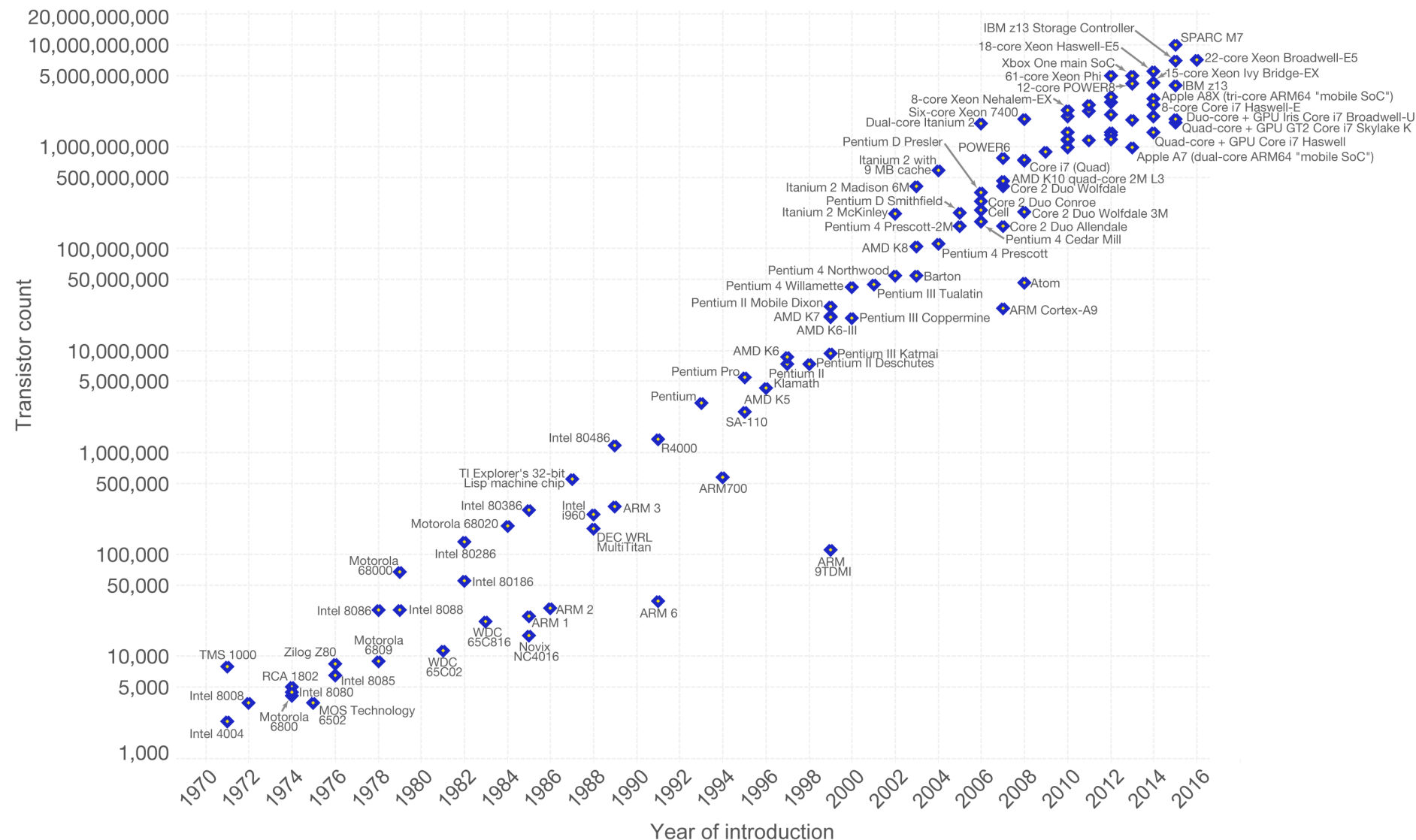
Microprocessor Transistor Counts 1971-2011 & Moore's Law



Number of transistors on an integrated circuit doubles ~ every two years

Our World
in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

The data visualization is available at [OurWorldinData.org](https://ourworldindata.org). There you find more visualizations and research on this topic.

Licensed under [CC-BY-SA](#) by the author Max Roser.

Recommended Reading

- Moore, “Cramming more components onto integrated circuits,” Electronics Magazine, 1965.
- Only 3 pages
- A quote:
"With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65 000 components on a single silicon chip."
- Another quote:
"Will it be possible to remove the heat generated by tens of thousands of components in a single silicon chip?"

How Do We Keep Moore's Law

- **Manufacturing smaller transistors/structures**
 - Some structures are already a few atoms in size
- **Developing materials with better properties**
 - Copper instead of Aluminum (better conductor)
 - Hafnium Oxide, air for Insulators
 - Making sure all materials are compatible is the challenge
- **Optimizing the manufacturing steps**
 - How to use 193nm ultraviolet light to pattern 20nm structures
- **New technologies**
 - FinFET, Gate All Around transistor, Single Electron Transistor...

Combinational Logic Circuits

We Can Now Build Logic Circuits

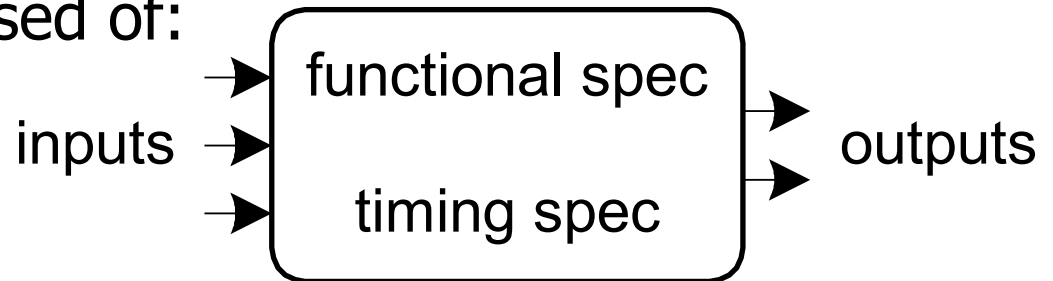
Now, we understand the workings of the basic logic gates

What is our next step?

Build some of the logic structures that are important components of the microarchitecture of a computer!

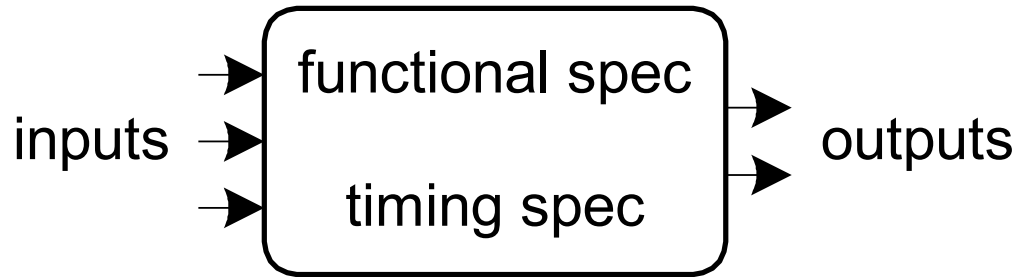
- A logic circuit is composed of:

- Inputs
- Outputs



- *Functional specification* (describes relationship between inputs and outputs)
- *Timing specification* (describes the delay between inputs changing and outputs responding)

Types of Logic Circuits



■ **Combinational Logic**

- ❑ Memoryless
- ❑ Outputs are strictly dependent on the combination of input values that are being applied to circuit *right now*
- ❑ In some books called Combinatorial Logic

■ **Later we will learn: Sequential Logic**

- ❑ Has memory
 - Structure stores history → Can "store" data values
- ❑ Outputs are determined by previous (historical) and current values of inputs

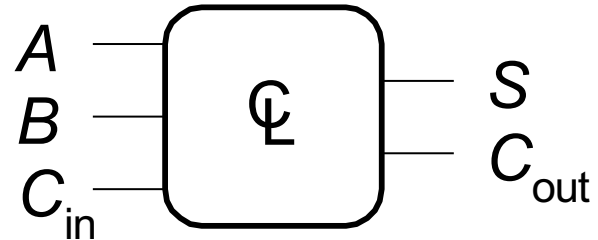
Boolean Equations

Functional Specification

- **Functional specification** of outputs in terms of inputs
- What do we mean by “function”?
 - Unique **mapping** from input values to output values
 - The **same** input values produce the **same** output value every time
 - **No memory** (does not depend on the history of input values)
- **Example (full 1-bit adder – more later):**

$$S = F(A, B, C_{in})$$

$$C_{out} = G(A, B, C_{in})$$

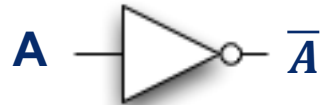


$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

Simple Equations: NOT / AND / OR

\bar{A} (reads “not A”) is 1 iff A is 0



A	\bar{A}
0	1
1	0

$A \cdot B$ (reads “A and B”) is 1 iff A and B are both 1



A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

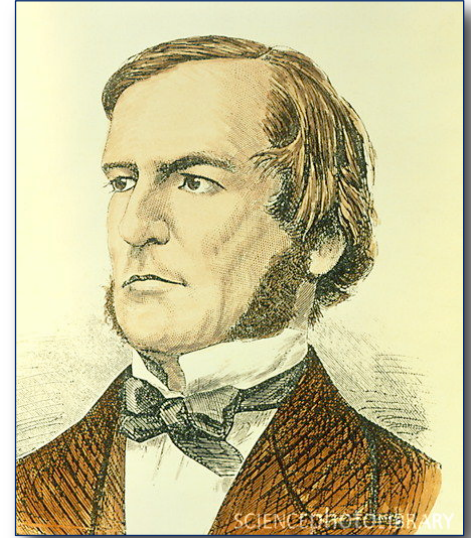
$A + B$ (reads “A or B”) is 1 iff either A or B is 1



A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

Boolean Algebra: Big Picture

- An algebra on 1's and 0's
 - with AND, OR, NOT operations
- What you start with
 - **Axioms:** basic things about objects and operations you just assume to be true at the start
- What you derive first
 - **Laws and theorems:** allow you to manipulate Boolean expressions
 - ...also allow us to do some simplification on Boolean expressions
- What you derive later
 - More "sophisticated" properties useful for manipulating digital designs represented in the form of Boolean equations



Boolean Algebra: Axioms

<i>Formal version</i>	<i>English version</i>
1. <i>B</i> contains at least two elements, <i>0</i> and <i>1</i> , such that $0 \neq 1$	Math formality...
2. <i>Closure</i> $a, b \in B$, (i) $a + b \in B$ (ii) $a \bullet b \in B$	Result of AND, OR stays in set you start with
3. <i>Commutative Laws</i> : $a, b \in B$, (i) (ii)	For primitive AND, OR of 2 inputs, order doesn't matter
4. <i>Identities</i> : $0, 1 \in B$ (i) (ii)	There are identity elements for AND, OR, that give you back what you started with
5. <i>Distributive Laws</i> : (i) (ii)	<ul style="list-style-type: none">• distributes over +, just like algebra ...but + distributes over •, also (!!)
6. <i>Complement</i> : (i) $\overline{}$ (ii)	There is a complement element; AND/ORing with it gives the identity elm.

Boolean Algebra: Duality

■ Observation

- All the axioms come in “dual” form
- Anything true for an expression also true for its dual
- So any derivation you could make that is true, can be flipped into dual form, and it stays true

■ Duality — More formally

- A dual of a Boolean expression is derived by replacing
 - Every AND operation with... an OR operation
 - Every OR operation with... an AND
 - Every constant 1 with... a constant 0
 - Every constant 0 with... a constant 1
 - But don't change any of the literals or play with the complements!

Example

$$\begin{aligned} a \cdot (b + c) &= (a \cdot b) + (a \cdot c) \\ \rightarrow a + (b \cdot c) &= (a + b) \cdot (a + c) \end{aligned}$$

Boolean Algebra: Useful Laws

Operations with 0 and 1:

1. $X + 0 = X$

2. $X + 1 = 1$

Dual
↓

1D. $X \cdot 1 = X$
2D. $X \cdot 0 = 0$

AND, OR with identities
gives you back the original
variable or the identity

Idempotent Law:

3. $X + X = X$

3D. $X \cdot X = X$

AND, OR with self = self

Involution Law:

4. $\overline{\overline{X}} = X$

double complement =
no complement

Laws of Complementarity:

5. $X + \overline{X} = 1$

5D. $X \cdot \overline{X} = 0$

AND, OR with complement
gives you an identity

Commutative Law:

6. $X + Y = Y + X$

6D. $X \cdot Y = Y \cdot X$

Just an axiom...

Useful Laws (cont)

Associative Laws:

$$\begin{aligned} 7. (X + Y) + Z &= X + (Y + Z) \\ &= X + Y + Z \end{aligned}$$

$$\begin{aligned} 7D. (X \cdot Y) \cdot Z &= X \cdot (Y \cdot Z) \\ &= X \cdot Y \cdot Z \end{aligned}$$

Parenthesis order
does not matter

Distributive Laws:

$$8. X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)$$

$$8D. X + (Y \cdot Z) = (X + Y) \cdot (X + Z) \quad \text{Axiom}$$

Simplification Theorems:

9.

9D.

10.

10D.

11.

11D.

Useful for
simplifying
expressions

Actually worth remembering — they show up a lot in real designs...

Boolean Algebra: Proving Things

Proving theorems via axioms of Boolean Algebra:

EX: Prove the theorem: $X \cdot Y + X \cdot \bar{Y} = X$

Distributive (5)

Complement (6)

Identity (4)

EX2: Prove the theorem: $X + X \cdot Y = X$

Identity (4)

Distributive (5)

Identity (2)

Identity (4)

DeMorgan's Law: Enabling Transformations

DeMorgan's Law:

$$12. \overline{(X + Y + Z + \dots)} = \bar{X} \cdot \bar{Y} \cdot \bar{Z} \cdot \dots$$

$$12D. \overline{(X \cdot Y \cdot Z \cdot \dots)} = \bar{X} + \bar{Y} + \bar{Z} + \dots$$

■ Think of this as a transformation

- Let's say we have:

$$F = A + B + C$$

- Applying DeMorgan's Law (12), gives us

$$F = \overline{\overline{(A + B + C)}} = \overline{(\bar{A} \cdot \bar{B} \cdot \bar{C})}$$

At least one of A, B, C is TRUE --> It is **not** the case that A, B, C are **all** false

DeMorgan's Law (Continued)

These are conversions between **different types of logic functions**
They can prove useful if you do not have **every type of gate**

$$A = \overline{(X + Y)} = \bar{X}\bar{Y}$$

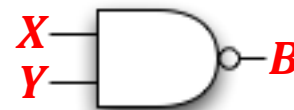
**NOR is equivalent to AND
with inputs complemented**



X	Y	$\overline{X + Y}$	\bar{X}	\bar{Y}	$\bar{X}\bar{Y}$
0	0	1	1	1	1
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	0

$$B = \overline{(XY)} = \bar{X} + \bar{Y}$$

**NAND is equivalent to OR
with inputs complemented**



X	Y	\overline{XY}	\bar{X}	\bar{Y}	$\bar{X} + \bar{Y}$
0	0	1	1	1	1
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	0	0	0

Digital Design & Computer Arch.

Lecture 4: Combinational Logic I

Prof. Onur Mutlu

ETH Zürich

Spring 2020

28 February 2020

We Did Not Cover the
Following Slides in Lecture 4

Using Boolean Equations to Represent a Logic Circuit

Sum of Products Form: Key Idea

- Assume we have the truth table of a Boolean Function
- How do we express the function in terms of the inputs in a **standard** manner?
- Idea: **Sum of Products** form
- Express the truth table as a two-level Boolean expression
 - that contains **all** input variable combinations that result in a 1 output
 - If ANY of the combinations of input variables that results in a 1 is TRUE, then the output is 1
 - $F = \text{OR of all input variable combinations that result in a 1}$

Some Definitions

- **Complement:** variable with a bar over it
 $\bar{A}, \bar{B}, \bar{C}$
- **Literal:** variable or its complement
 $A, \bar{A}, B, \bar{B}, C, \bar{C}$
- **Implicant:** product (AND) of literals
 $(A \cdot B \cdot \bar{C}), (\bar{A} \cdot C), (B \cdot \bar{C})$
- **Minterm:** product (AND) that includes **all** input variables
 $(A \cdot B \cdot \bar{C}), (\bar{A} \cdot \bar{B} \cdot C), (\bar{A} \cdot B \cdot \bar{C})$
- **Maxterm:** sum (OR) that includes **all** input variables
 $(A + \bar{B} + \bar{C}), (\bar{A} + B + \bar{C}), (A + B + \bar{C})$

Two-Level Canonical (Standard) Forms

- **Truth table** is the unique **signature** of a Boolean *function* ...
 - But, it is an expensive representation
- A Boolean function can have many alternative Boolean expressions
 - i.e., many alternative Boolean expressions (and gate realizations) may have the same truth table (and function)
- **Canonical** form: **standard form for a Boolean expression**
 - Provides a unique algebraic signature
 - If they all say the same thing, why do we care?
 - Different Boolean expressions lead to different gate realizations

Two-Level Canonical Forms

Sum of Products Form (SOP)

Also known as **disjunctive normal form** or **minterm expansion**

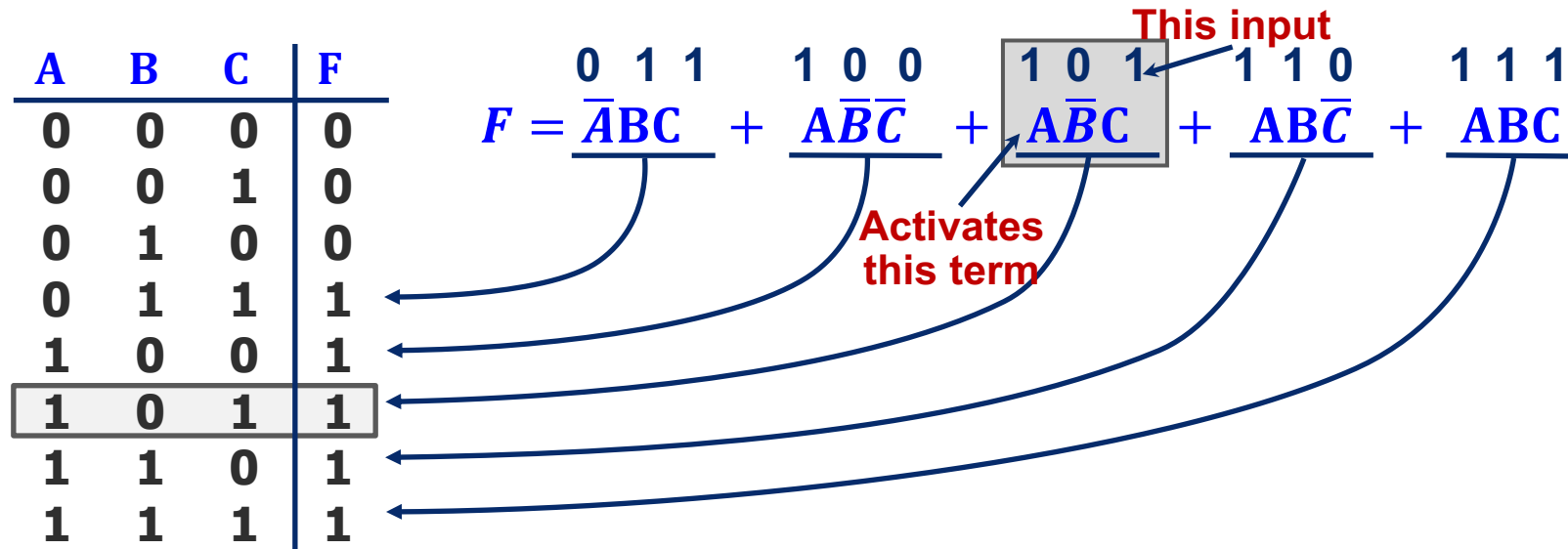
A	B	C	F	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\bar{A}BC$
1	0	0	1	$A\bar{B}\bar{C}$
1	0	1	1	$A\bar{B}C$
1	1	0	1	$AB\bar{C}$
1	1	1	1	ABC

$F = \bar{A}BC + A\bar{B}\bar{C} + A\bar{B}C + AB\bar{C} + ABC$

- Each row in a truth table has a minterm
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)

All Boolean equations can be written in SOP form

SOP Form — Why Does It Work?



- Only the shaded product term — $\bar{A}\bar{B}C = 1 \cdot 0 \cdot 1$ — will be 1
- No other product terms will “turn on” — they will all be 0
- So if inputs A B C correspond to a product term in expression,
 - We get $0 + 0 + \dots + 1 + \dots + 0 + 0 = 1$ for output
- If inputs A B C do not correspond to any product term in expression
 - We get $0 + 0 + \dots + 0 = 0$ for output

Aside: Notation for SOP

- Standard “shorthand” notation
 - If we agree on the **order** of the variables in the rows of truth table...
 - then we can enumerate each row with the decimal number that corresponds to the binary number created by the input pattern

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

100 = decimal 4 so this is minterm #4, or m4

111 = decimal 7 so this is minterm #7, or m7

f =

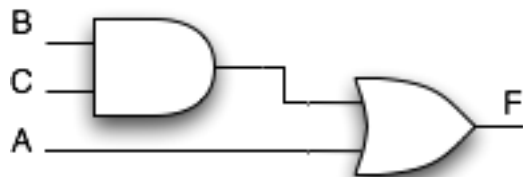
We can write this as a sum of products

Or, we can use a summation notation

Canonical SOP Forms

A	B	C	minterms
0	0	0	$\overline{A}\overline{B}\overline{C} = m_0$
0	0	1	$\overline{A}\overline{B}C = m_1$
0	1	0	$\overline{A}B\overline{C} = m_2$
0	1	1	$\overline{A}BC = m_3$
1	0	0	$A\overline{B}\overline{C} = m_4$
1	0	1	$A\overline{B}C = m_5$
1	1	0	$AB\overline{C} = m_6$
1	1	1	$ABC = m_7$

Shorthand Notation for
Minterms of 3 Variables



2-Level AND/OR
Realization

F in canonical form:

$$F(A,B,C) = \sum m(3,4,5,6,7) \\ = m_3 + m_4 + m_5 + m_6 + m_7$$

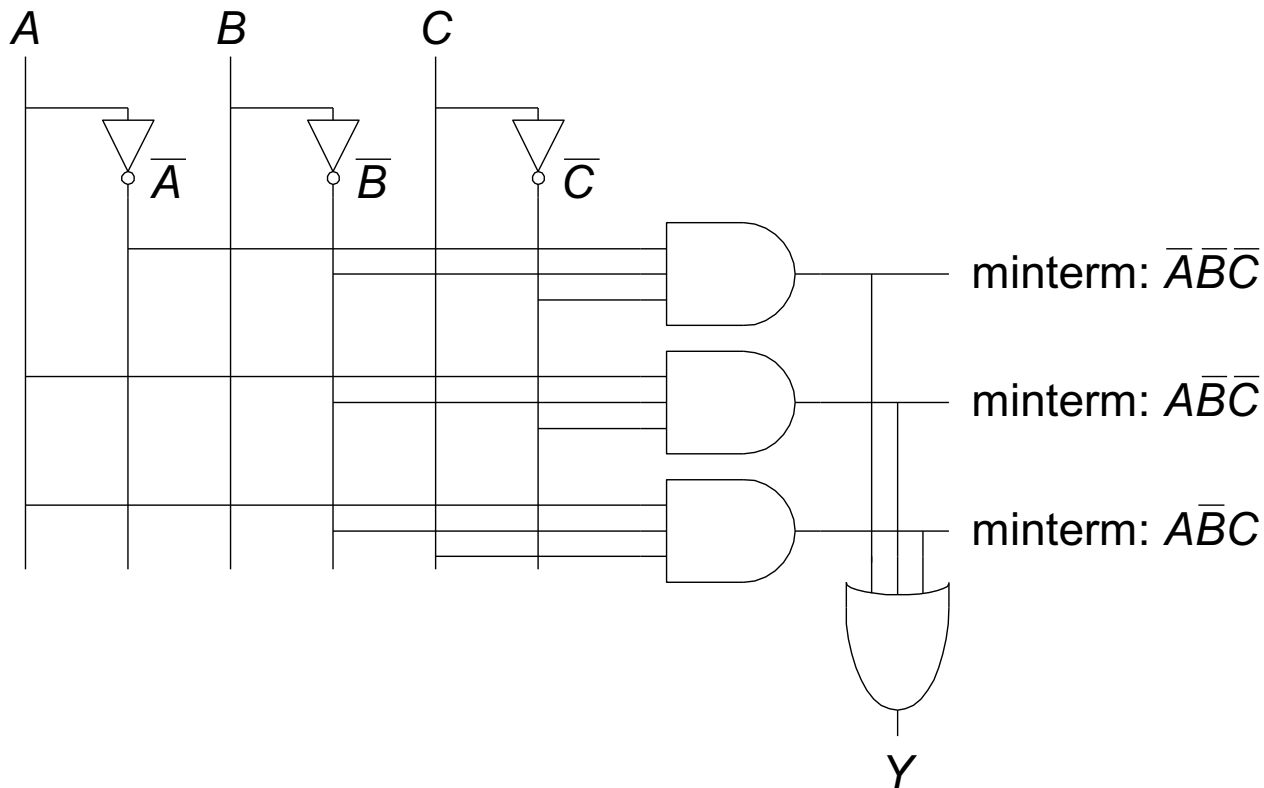
F =

canonical form \neq minimal form

F

From Logic to Gates

- **SOP (sum-of-products) leads to two-level logic**
- Example: $Y = (\bar{A} \cdot \bar{B} \cdot \bar{C}) + (A \cdot \bar{B} \cdot \bar{C}) + (A \cdot \bar{B} \cdot C)$



Alternative Canonical Form: POS

We can have another form of representation

DeMorgan of SOP of \bar{F}

A product of sums (POS)

$$F = (A + B + C)(A + B + \bar{C})(A + \bar{B} + C)$$

products

sums

Each sum term represents one of the “**zeros**” of the function

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$F = \overset{0}{(A + B + C)} \overset{0}{(A + B + \bar{C})} \overset{1}{(A + \bar{B} + C)}$

This input

Activates this term

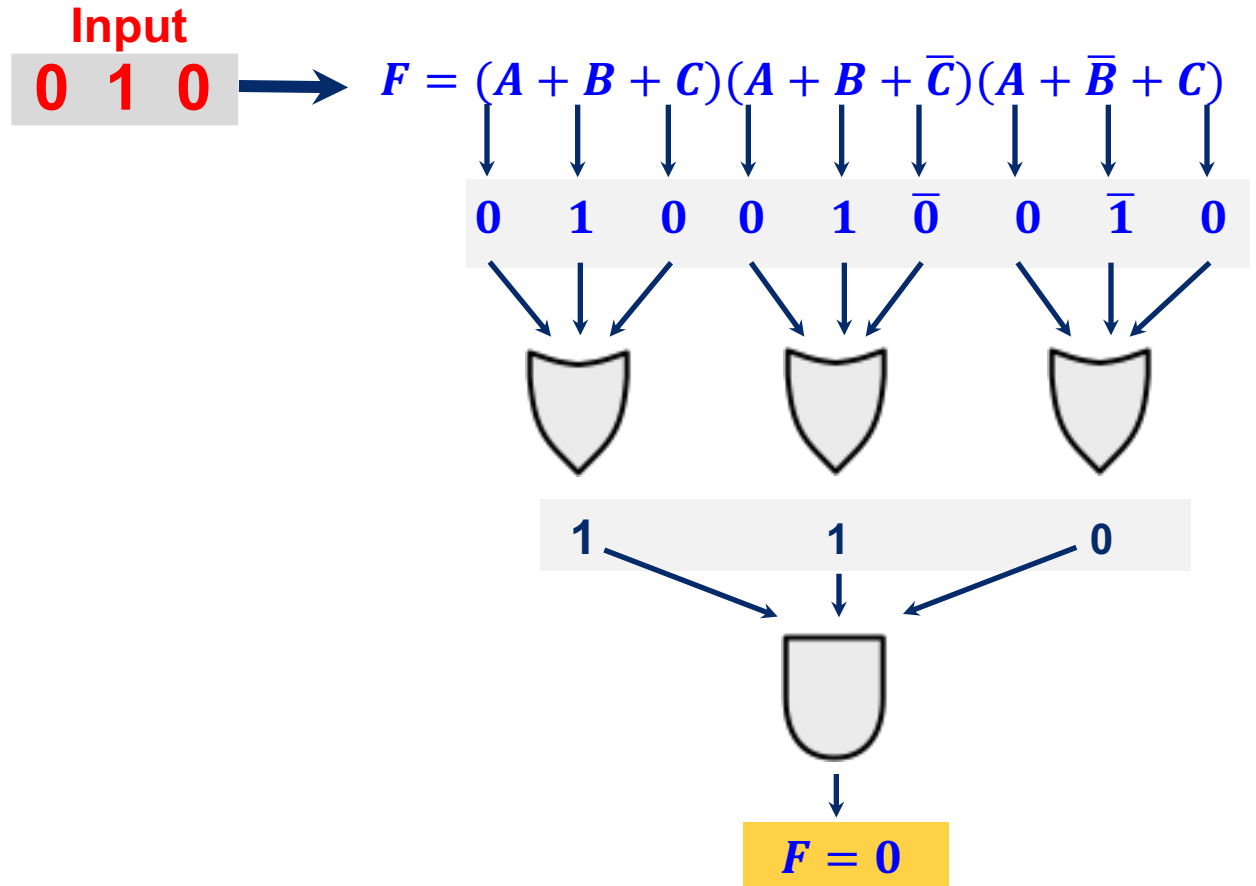
For the given input, only the shaded sum term will equal 0

$$A + \bar{B} + C = 0 + \bar{1} + 0$$

Anything ANDed with 0 is 0; Output F will be 0

Consider $A=0, B=1, C=0$

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



Only one of the products will be 0, anything ANDed with 0 is 0

Therefore, the output is $F = 0$

POS: How to Write It

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$F = (A + B + C)(A + B + \bar{C})(A + \bar{B} + C)$

$A \quad \bar{B} \quad C$

$A + \bar{B} + C$

Maxterm form:

1. Find truth table rows where F is 0
2. 0 in input col → true literal
3. 1 in input col → complemented literal
4. OR the literals to get a Maxterm
5. AND together all the Maxterms

Or just remember, POS of F is the same as the DeMorgan of SOP of \bar{F} !!

Canonical POS Forms

Product of Sums / Conjunctive Normal Form / Maxterm Expansion

$$F = (A + B + C)(A + B + \bar{C})(A + \bar{B} + C)$$

$$\prod M(0, 1, 2)$$

A	B	C	Maxterms
0	0	0	$A + B + C = M0$
0	0	1	$A + B + \bar{C} = M1$
0	1	0	$A + \bar{B} + C = M2$
0	1	1	$A + \bar{B} + \bar{C} = M3$
1	0	0	$\bar{A} + B + C = M4$
1	0	1	$\bar{A} + B + \bar{C} = M5$
1	1	0	$\bar{A} + \bar{B} + C = M6$
1	1	1	$\bar{A} + \bar{B} + \bar{C} = M7$

Maxterm shorthand notation
for a function of three variables

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Note that you
form the
maxterms around
the “zeros” of the
function

This is **not** the
complement of
the function!

Useful Conversions

1. **Minterm to Maxterm conversion:**

rewrite minterm shorthand using maxterm shorthand
replace minterm indices with the indices not already used

E.g., $F(A, B, C) = \sum m(3, 4, 5, 6, 7) = \prod M(0, 1, 2)$

2. **Maxterm to Minterm conversion:**

rewrite maxterm shorthand using minterm shorthand
replace maxterm indices with the indices not already used

E.g., $F(A, B, C) = \prod M(0, 1, 2) = \sum m(3, 4, 5, 6, 7)$

3. **Expansion of F to expansion of \bar{F} :**

$$\begin{array}{ll} \text{E. g., } F(A, B, C) = \sum m(3, 4, 5, 6, 7) & \longrightarrow \bar{F}(A, B, C) = \sum m(0, 1, 2) \\ = \prod M(0, 1, 2) & \longrightarrow = \prod M(3, 4, 5, 6, 7) \end{array}$$

4. **Minterm expansion of F to Maxterm expansion of \bar{F} :**

rewrite in Maxterm form, using the same indices as F

$$\begin{array}{ll} \text{E. g., } F(A, B, C) = \sum m(3, 4, 5, 6, 7) & \longrightarrow \bar{F}(A, B, C) = \prod M(3, 4, 5, 6, 7) \\ = \prod M(0, 1, 2) & \longrightarrow = \sum m(0, 1, 2) \end{array}$$

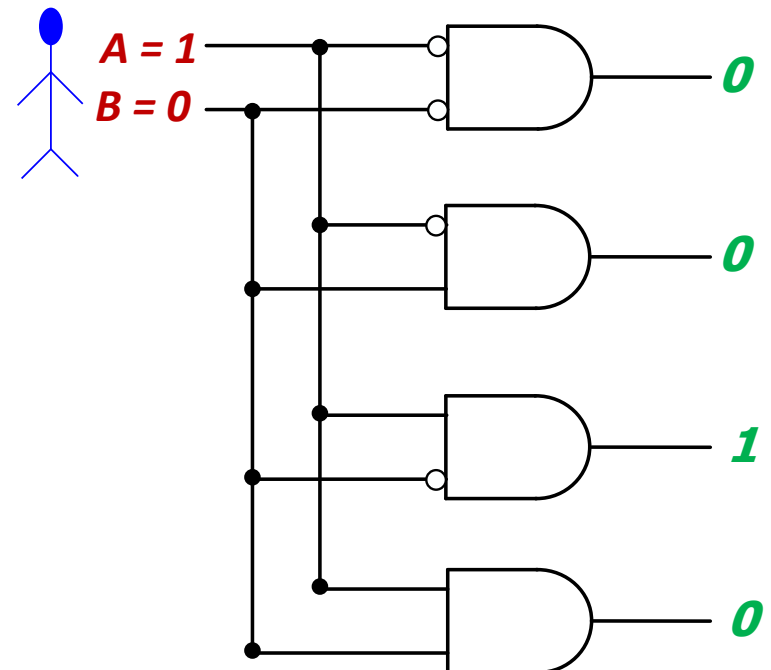
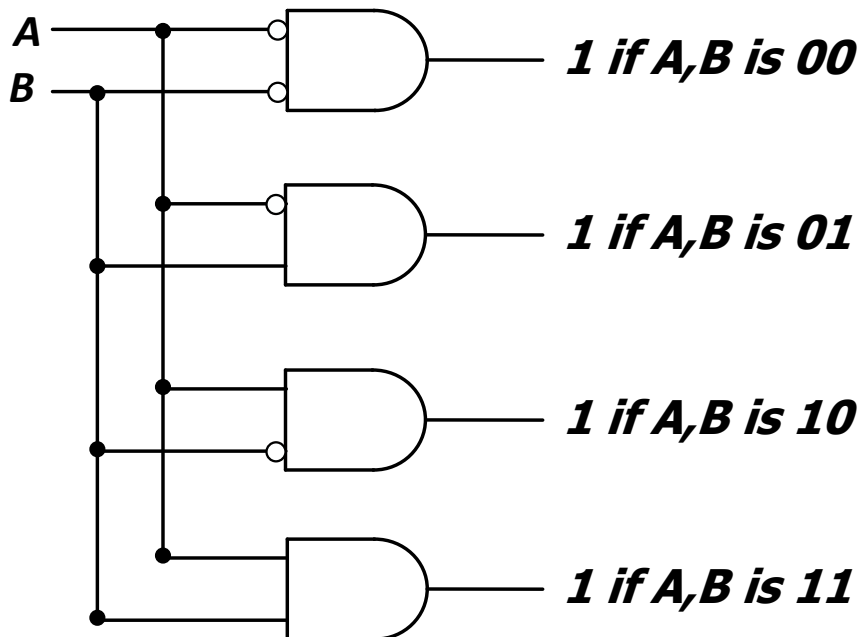
Combinational Building Blocks used in Modern Computers

Combinational Building Blocks

- Combinational logic is often grouped into larger building blocks to build more **complex systems**
 - Hides the **unnecessary gate-level details** to emphasize the function of the building block
 - We now look at:
 - Decoders
 - Multiplexers
 - Full adder
 - PLA (Programmable Logic Array)
-

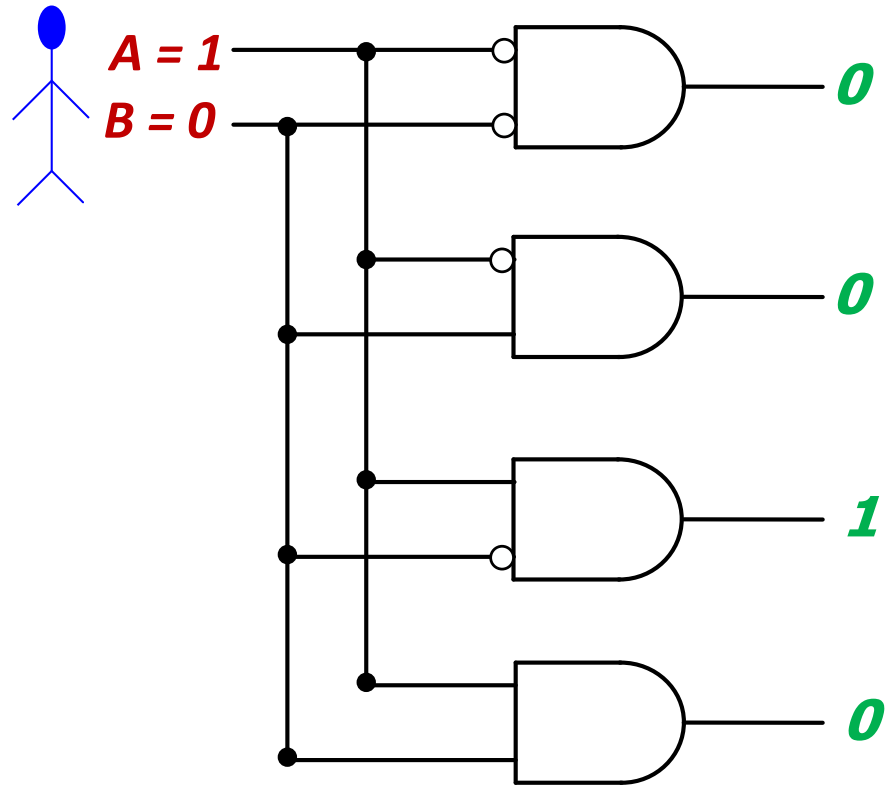
Decoder

- n inputs and 2^n outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The **one output** that is logically 1 is the output corresponding to the input **pattern** that the logic circuit is expected to detect



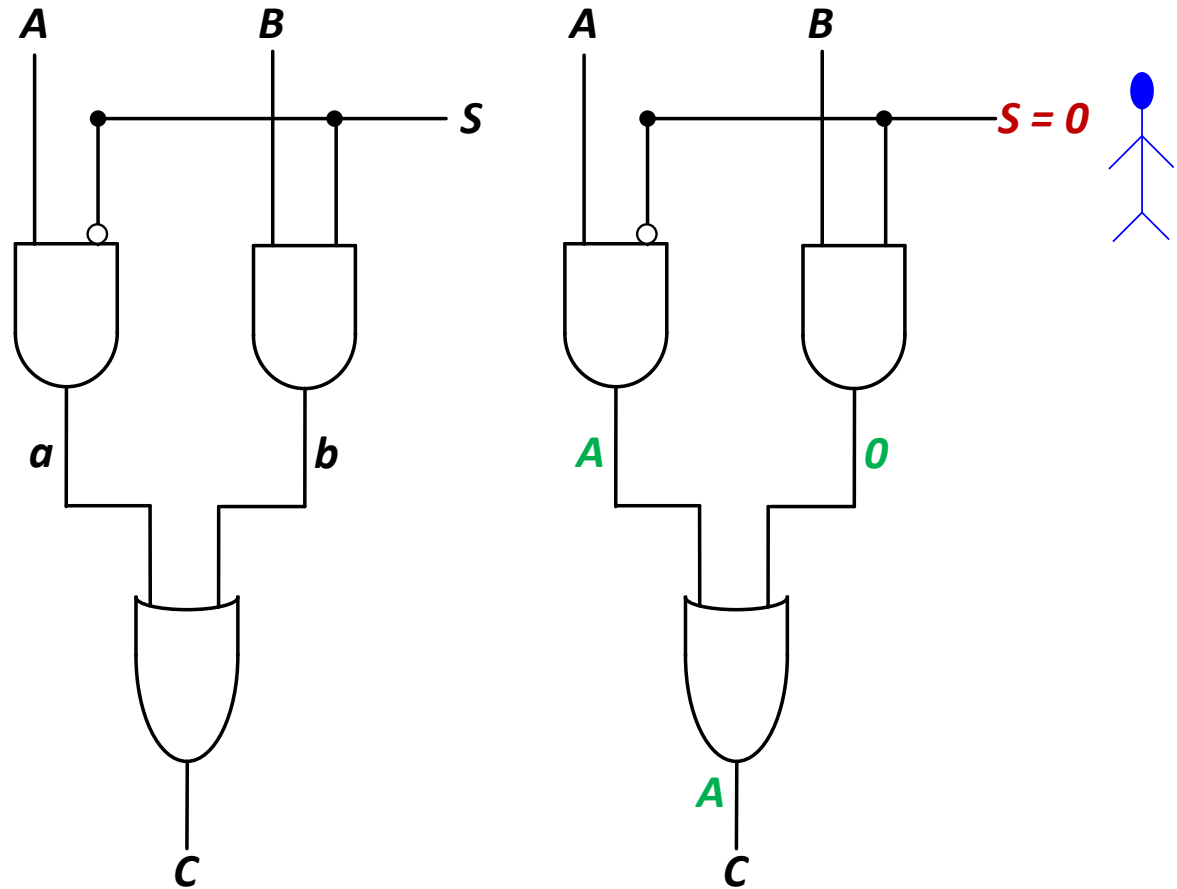
Decoder

- The decoder is useful in determining how to interpret a bit pattern
 - **It could be the address of a row in DRAM, that the processor intends to read from**
 - **It could be an instruction in the program and the processor has to decide what action to do! (based on *instruction opcode*)**



Multiplexer (MUX), or Selector

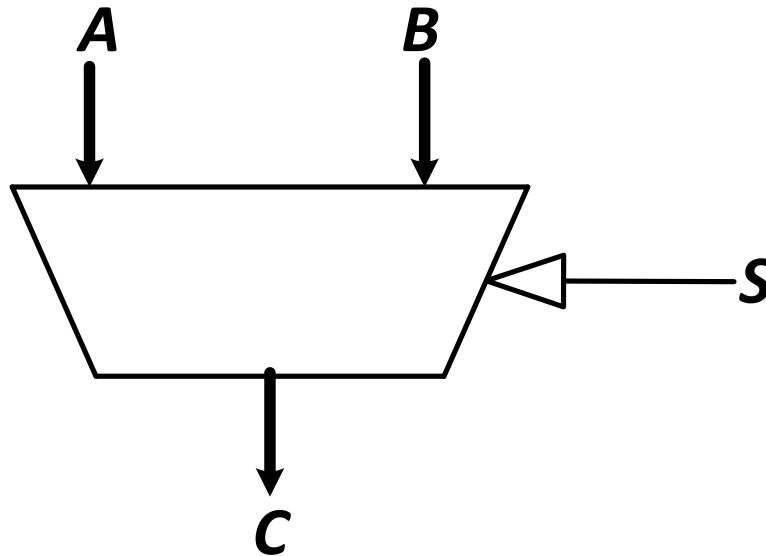
- **Selects** one of the N inputs to connect it to the output
- Needs $\log_2 N$ -bit control input
- 2:1 MUX



Multiplexer (MUX)

- The output C is always connected to either the input A or the input B
 - Output value depends on the value of the **select line S**

S	C
0	A
1	B



- **Your task:** Draw the schematic for an 8-input (8:1) MUX
 - Gate level: as a combination of basic AND, OR, NOT gates
 - Module level: As a combination of 2-input (2:1) MUXes

Full Adder (I)

■ Binary addition

- Similar to decimal addition
- From right to left
- One column at a time
- One sum and one carry bit

$$\begin{array}{r}
 a_{n-1}a_{n-2} \dots a_1a_0 \\
 b_{n-1}b_{n-2} \dots b_1b_0 \\
 \underline{C_n C_{n-1} \dots C_1} \\
 S_{n-1} \dots S_1S_0
 \end{array}$$

↓

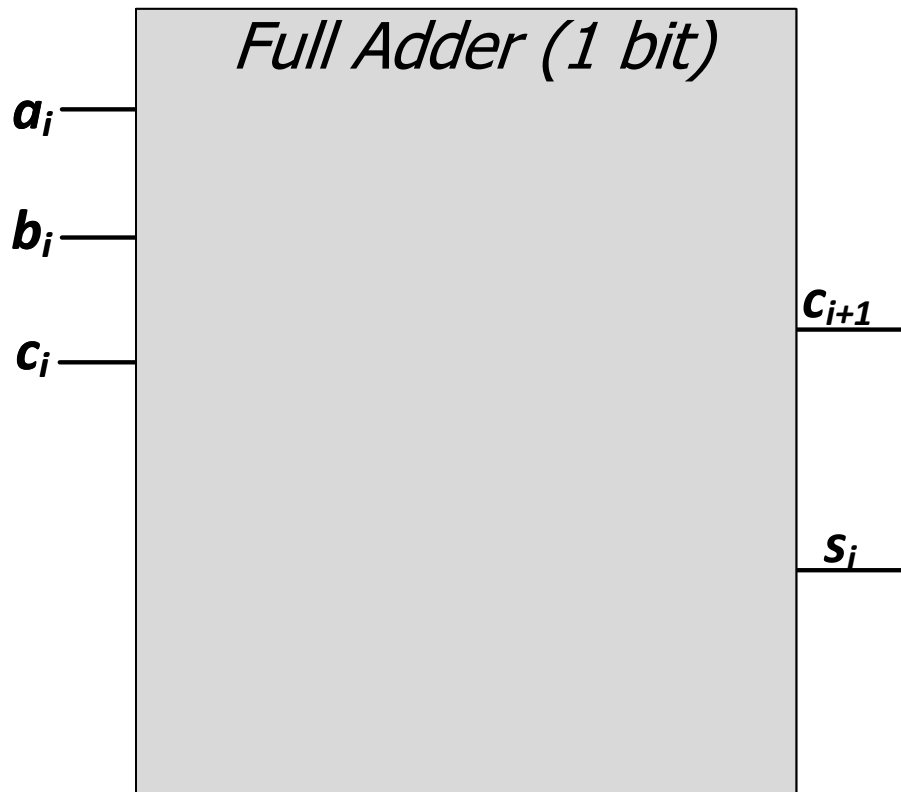
- Truth table of binary addition on **one column** of bits within two n-bit operands

a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder (II)

■ Binary addition

- N 1-bit additions
- **SOP of 1-bit addition**



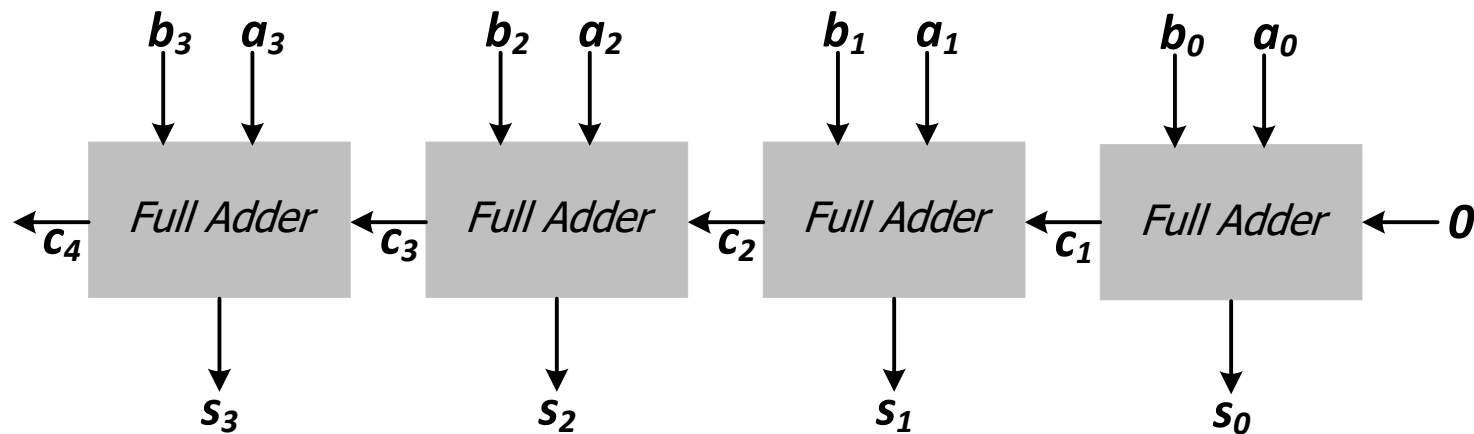
$$\begin{array}{r} a_{n-1}a_{n-2} \dots a_1a_0 \\ b_{n-1}b_{n-2} \dots b_1b_0 \\ \hline C_n C_{n-1} \dots C_1 \\ \hline S_{n-1} \dots S_1S_0 \end{array}$$

A vertical arrow points downwards from the right side of the addition, indicating the progression of bits from a_0 to a_{n-1} .

a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

4-Bit Adder from Full Adders

- Creating a **4-bit adder** out of 1-bit full adders
 - To add two 4-bit binary numbers A and B



$$\begin{array}{rcccc} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline c_4 & c_3 & c_2 & c_1 & \\ \hline s_3 & s_2 & s_1 & s_0 & \end{array}$$

$$\begin{array}{rcccc} & 1 & 0 & 1 & 1 \\ + & 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 1 & 1 & \\ \hline 0 & 1 & 0 & 0 & \end{array}$$

The Programmable Logic Array (PLA)

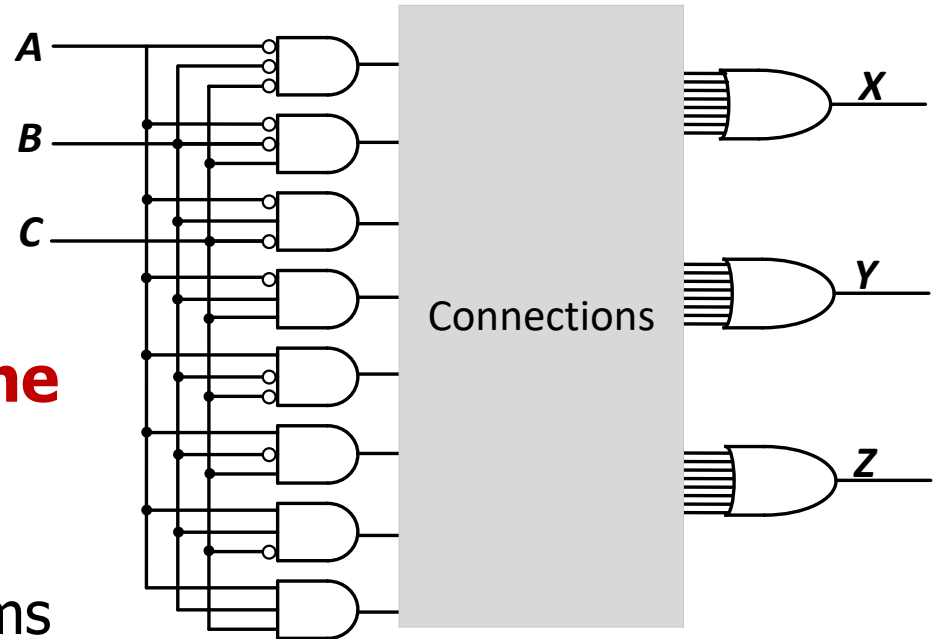
- The below logic structure is a very **common** building block for implementing any collection of logic functions one wishes to

- An **array** of AND gates followed by an **array** of OR gates
- **How do we determine the number of AND gates?**

- **Remember SOP:** the number of possible minterms

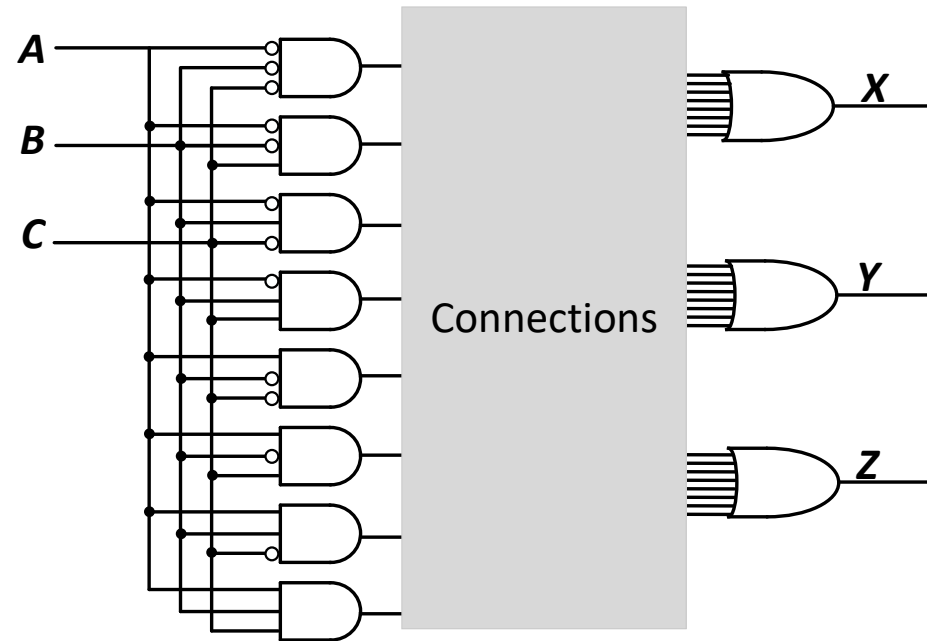
- For an n -input logic function, we need a PLA with 2^n n -input AND gates

- **How do we determine the number of OR gates?** The number of output columns in the truth table

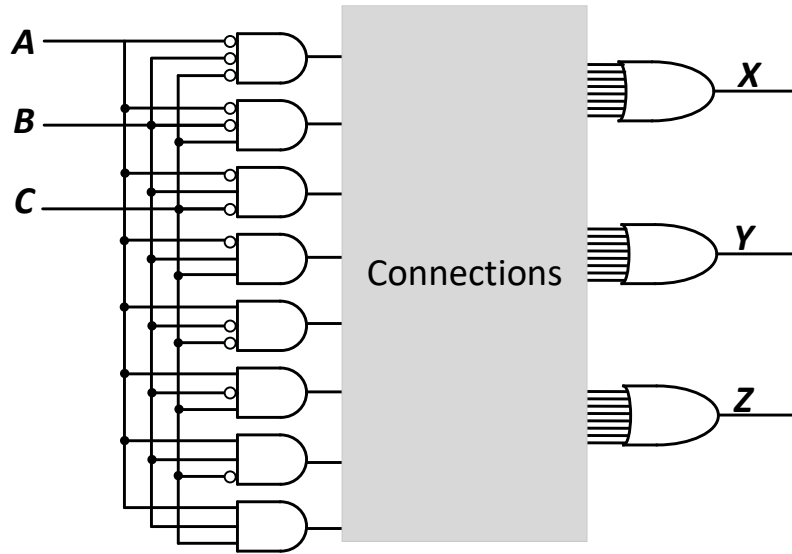


The Programmable Logic Array (PLA)

- How do we implement a logic function?
 - Connect the output of an AND gate to the input of an OR gate if the corresponding minterm is included in the SOP
 - This is a simple programmable logic
- **Programming a PLA:** we program the connections from AND gate outputs to OR gate inputs to implement a desired logic function
- Have you seen any other type of programmable logic?
 - Yes! An FPGA...
 - An FPGA uses more advanced structures, as we saw in Lecture 3

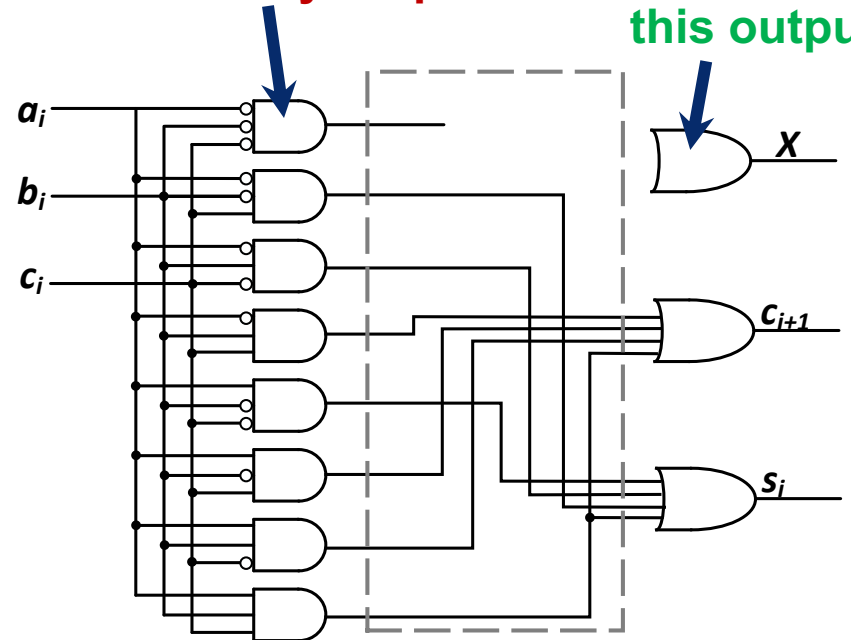


Implementing a Full Adder Using a PLA



This input should not be connected to any outputs

We do not need this output



Truth table of a full adder

a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Logical (Functional) Completeness

- **Any logic function** we wish to implement could be accomplished with a PLA
 - PLA consists of **only** AND gates, OR gates, and inverters
 - We just have to program connections based on SOP of the intended logic function
- The set of gates {AND, OR, NOT} is **logically complete** because we can build a circuit to carry out the specification of **any truth table** we wish, without using any other kind of gate
- NAND is also logically complete. So is NOR.
 - **Your task:** Prove this.

More Combinational Building Blocks

- H&H Chapter 2 in full
 - Required Reading
 - E.g., see Tri-state Buffer and Z values in Section 2.6
- H&H Chapter 5
 - Will be required reading soon.
- You will benefit greatly by reading the “combinational” parts of Chapter 5 soon.
 - Sections 5.1 and 5.2

Tri-State Buffer

- A tri-state buffer enables gating of different signals onto a wire

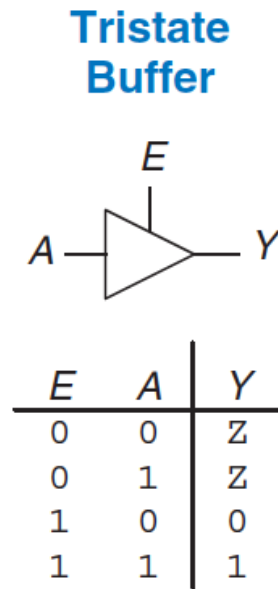


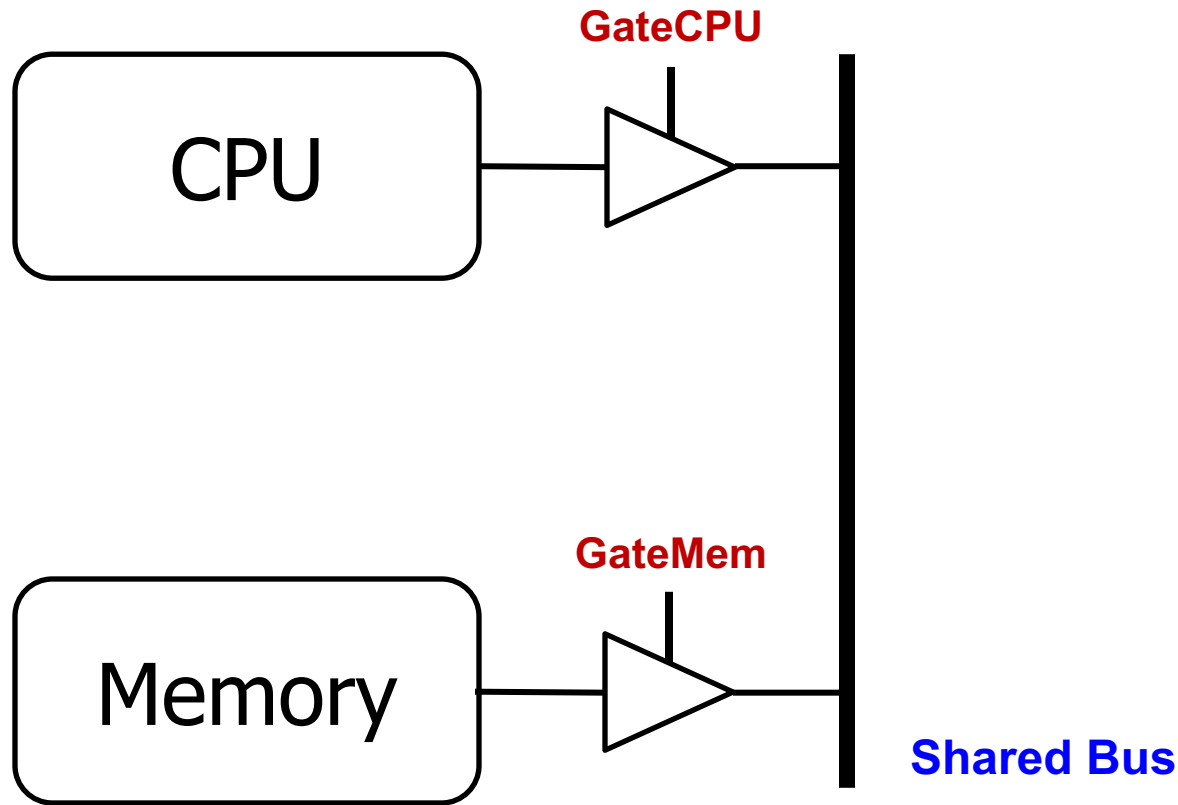
Figure 2.40 Tristate buffer

- **Floating signal (Z):** Signal that is not driven by any circuit
 - Open circuit, floating wire

Example: Use of Tri-State Buffers

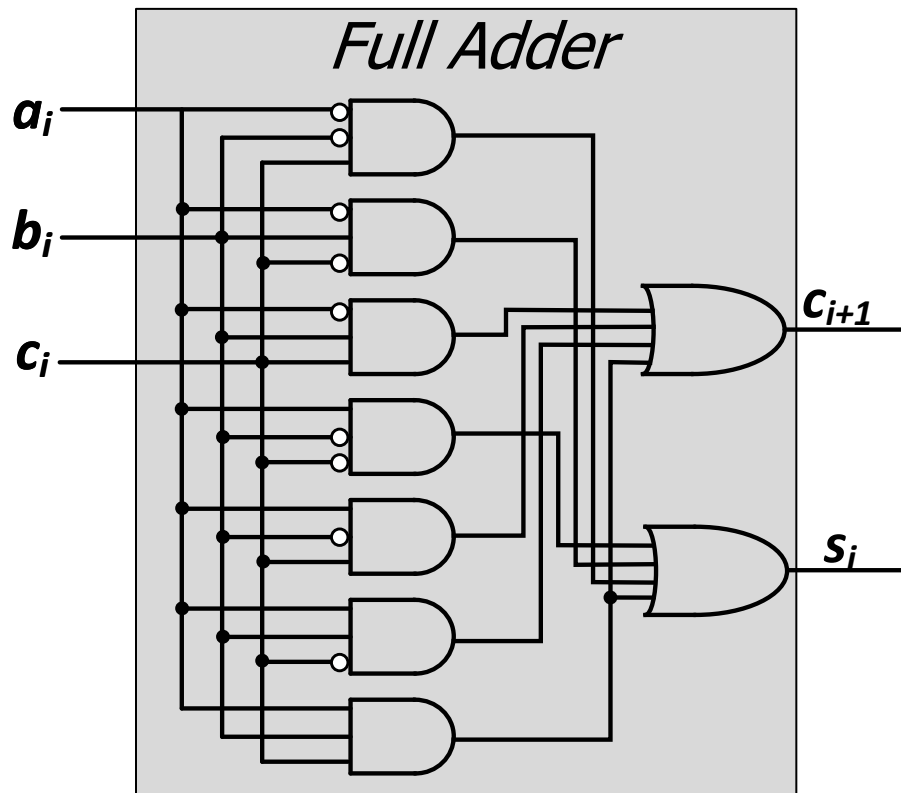
- Imagine a wire connecting the CPU and memory
 - At any time only the CPU or the memory can place a value on the wire, both not both
 - You can have two tri-state buffers: one driven by CPU, the other memory; and ensure at most one is enabled at any time

Example Design with Tri-State Buffers



Logic Simplification: Karnaugh Maps (K-Maps)

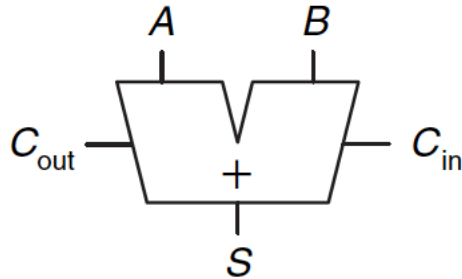
Recall: Full Adder in SOP Form Logic



a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Goal: Simplified Full Adder

Full
Adder



$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

How do we simplify Boolean logic?

Quick Recap on Logic Simplification

- The original Boolean expression (i.e., logic circuit) may not be optimal

$$F = \sim A(A + B) + (B + AA)(A + \sim B)$$

- Can we reduce a given Boolean expression to an equivalent expression **with fewer terms**?

$$F = A + B$$

- The **goal** of logic simplification:
 - **Reduce** the number of gates/inputs
 - **Reduce** implementation cost

A basis for what the automated design tools are doing today

Logic Simplification

■ Systematic techniques for simplifications

- amenable to automation

Key Tool: The Uniting Theorem — $F = A\bar{B} + AB$

A	B	F
0	0	0
0	1	0
1	0	1
1	1	1

$$F = A\bar{B} + AB = A(\bar{B} + B) = A(1) = A$$

Essence of Simplification:

Find two element subsets of the ON-set where only one variable changes its value. This single varying variable *can be eliminated!*

value is not needed

→ *B is eliminated, A remains*

A	B	G
0	0	1
0	1	0
1	0	1
1	1	0

$$G = \bar{A}\bar{B} + A\bar{B} = (\bar{A} + A)\bar{B} = \bar{B}$$

B's value stays the same within the ON-set rows

A's value changes within the ON-set rows

→ *A is eliminated, B remains*

Complex Cases

■ One example

$$C_{out} = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

■ Problem

- Easy to see how to apply Uniting Theorem...
- Hard to know if you applied it in all the right places...
- ...especially in a function of many more variables

■ Question

- Is there an easier way to find potential simplifications?
- i.e., potential applications of Uniting Theorem...?

■ Answer

- Need an intrinsically *geometric* representation for Boolean $f()$
- Something we can draw, see...

Karnaugh Map

- Karnaugh Map (K-map) method
 - K-map is an alternative method of representing the **truth table** that helps **visualize adjacencies** in up to 6 dimensions
 - Physical adjacency \leftrightarrow Logical adjacency

2-variable K-map

$A \backslash B$	0	1
0	00	01
1	10	11

3-variable K-map

$A \backslash BC$	00	01	11	10
0	000	001	011	010
1	100	101	111	110

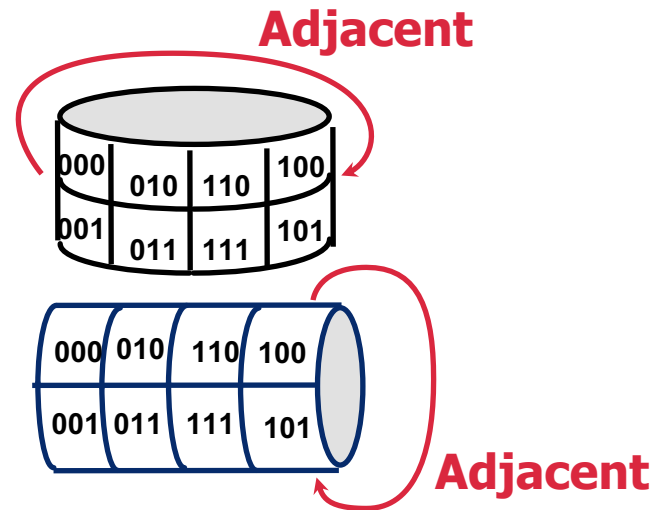
4-variable K-map

$AB \backslash CD$	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

Numbering Scheme: 00, 01, 11, 10 is called a “Gray Code” — only a *single bit (variable) changes* from one code word and the next code word

Karnaugh Map Methods

<i>A</i> \ <i>BC</i>	00	01	11	10
0	000	001	011	010
1	100	101	111	110



K-map adjacencies go “around the edges”
Wrap around from first to last column
Wrap around from top row to bottom row

K-map Cover - 4 Input Variables

$CD \backslash AB$	00	01	11	10
00	1	0	0	1
01	0	1	0	0
11	1	1	1	1
10	1	1	1	1

$$F(A, B, C, D) = \sum m(0, 2, 5, 8, 9, 10, 11, 12, 13, 14, 15)$$

$$F = A + \bar{B}\bar{D} + B\bar{C}D$$

Strategy for “circling” rectangles on Kmap:

Biggest “oops!” that people forget:

Logic Minimization Using K-Maps

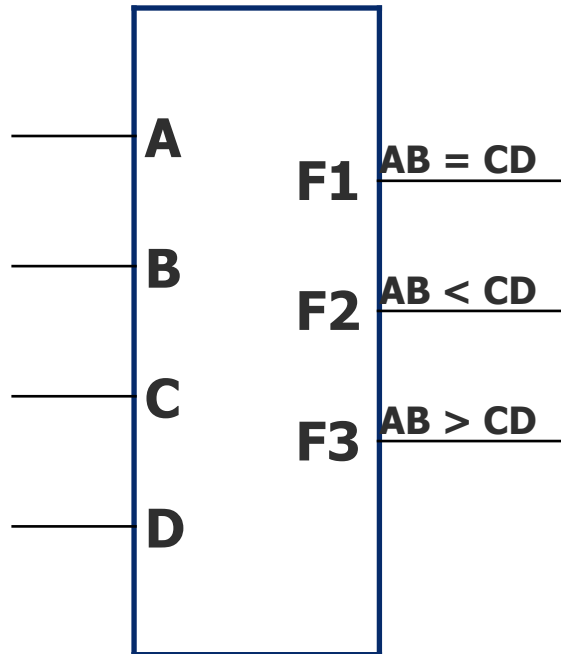
- Very simple guideline:
 - Circle all the rectangular blocks of 1's in the map, using the fewest possible number of circles
 - Each circle should be as large as possible
 - Read off the implicants that were circled

- More formally:
 - A Boolean equation is minimized when it is written as a sum of the fewest number of prime implicants
 - Each circle on the K-map represents an implicant
 - The largest possible circles are prime implicants

K-map Rules

- **What can be legally combined (circled) in the K-map?**
 - Rectangular groups of size 2^k for any integer k
 - Each cell has the same value (1, for now)
 - All values must be adjacent
 - Wrap-around edge is okay
- **How does a group become a term in an expression?**
 - Determine which literals are constant, and which vary across group
 - Eliminate varying literals, then AND the constant literals
 - constant 1 → use X , constant 0 → use \bar{X}
- **What is a good solution?**
 - Biggest groupings → eliminate more variables (literals) in each term
 - Fewest groupings → fewer terms (gates) all together
 - OR together all AND terms you create from individual groups

K-map Example: Two-bit Comparator



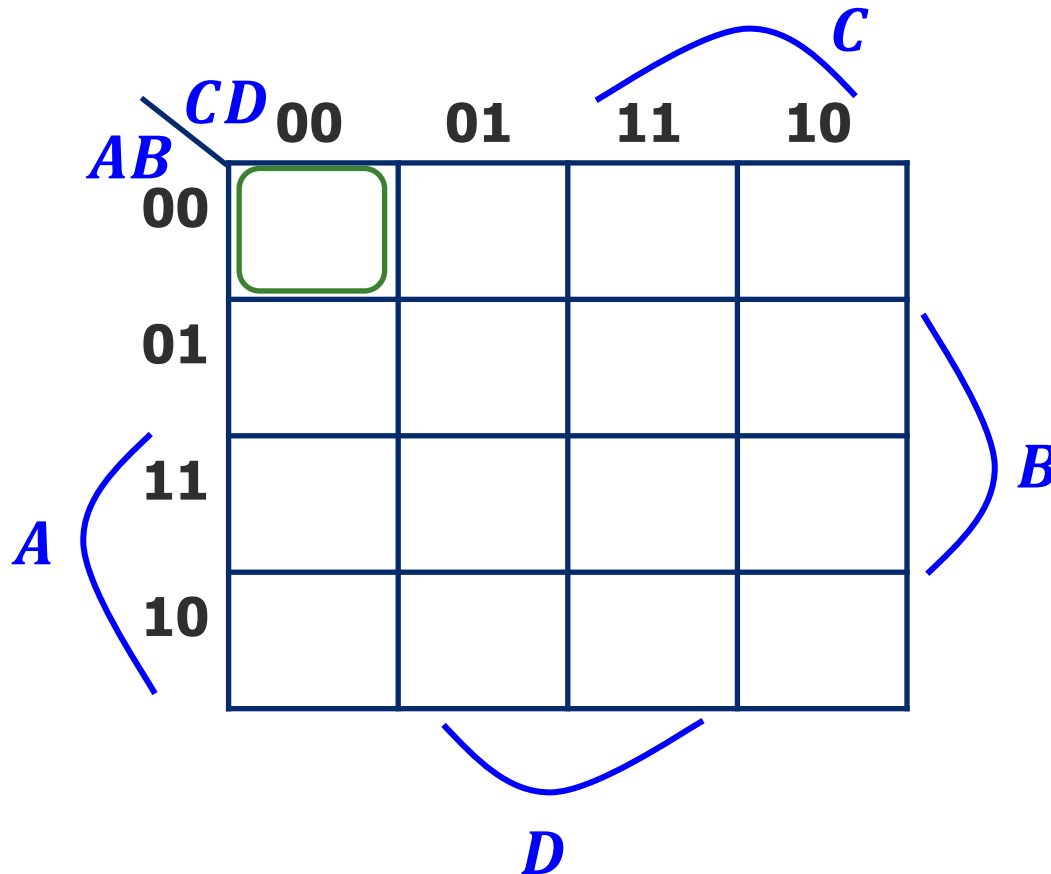
Design Approach:

Write a 4-Variable K-map
for each of the 3
output functions

A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

K-map Example: Two-bit Comparator (2)

K-map for F1

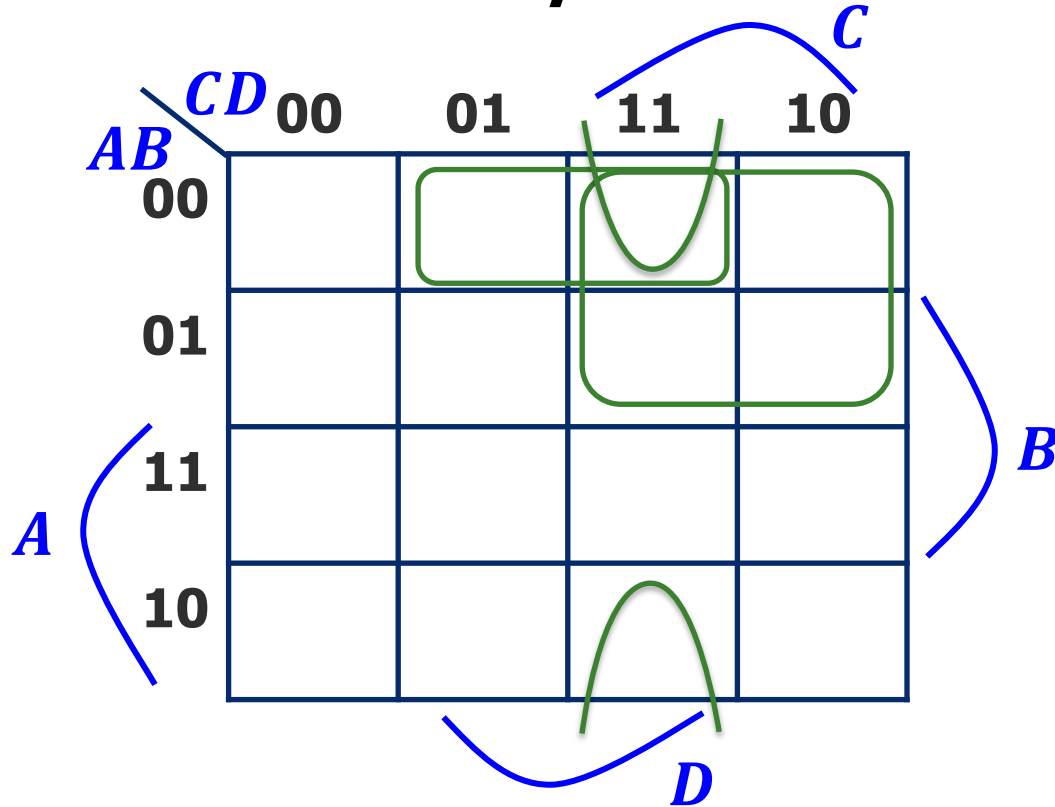


F1 =

A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

K-map Example: Two-bit Comparator (3)

K-map for F2



F2 =

F3 = ? (Exercise for you)

A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

K-maps with “Don’t Care”

- **Don’t Care** really means *I don’t care what my circuit outputs if this appears as input*
 - You have an engineering choice to use DON’T CARE patterns intelligently as 1 or 0 to better **simplify** the circuit

A	B	C	D	F	G
...					
0	1	1	0	X	X
0	1	1	1		
1	0	0	0	X	X
1	0	0	1		
...					

I can pick 00, 01, 10, 11 independently of below

I can pick 00, 01, 10, 11 independently of above

Example: BCD Increment Function

- BCD (Binary Coded Decimal) digits
 - Encode decimal digits 0 - 9 with bit patterns 0000_2 — 1001_2
 - When **incremented**, the decimal sequence is 0, 1, ..., 8, 9, 0, 1

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

These input patterns **should never be encountered** in practice (hey -- it's a BCD number!)
So, associated output values are **"Don't Cares"**

K-map for BCD Increment Function

A B

+

W X

Z (without don't cares) =

Z (with don't cares) =

10

1

X

X

10

X

X

Y

CD
AB

	00	01	11	10
00		1		1
01		1		1
11	X	X	X	X
10			X	X

Z

CD
AB

	00	01	11	10
00	1			1
01	1			1
11	X	X	X	X
10	1		X	X

Groupings: A (vertical, columns 00 and 10), B (vertical, columns 11 and 10), C (horizontal, rows 00 and 01), D (horizontal, rows 10 and 11).

K-map Summary

- **Karnaugh maps** as a formal systematic approach for logic simplification
- 2-, 3-, 4-variable K-maps
- K-maps with “**Don't Care**” outputs
- H&H Section 2.7

Digital Design & Computer Arch.

Lecture 4: Combinational Logic I

Prof. Onur Mutlu

ETH Zürich

Spring 2020

28 February 2020